

# NM-SESES

Multiphysics Software for  
Computer Aided Engineering

## User Manual

Version September 2012

Written by:

G. Sartoris

NM Numerical Modelling GmbH

Böhnirainstrasse 12

CH-8800 Thalwil

Switzerland

<http://www.nmtec.ch>

[mail://info@nmtec.ch](mailto:info@nmtec.ch)

Copyright, 1996 by **NM** Numerical Modelling GmbH.

This work is subject to copyright. All rights are reserved, whether the whole or part of the material is concerned, specifically the rights of reprinting, reuse of illustrations, translation, broadcasting, reproduction or storage in data banks. Duplication of this publication or parts thereof is permitted in connection with reviews, personal or scholarly usage. Permission for use must be obtained, in writing, from **NM**.

**NM** reserves the right to make changes in specifications at any time without notice. The information furnished by **NM** in this publication is believed to be accurate, however no responsibility is assumed for its use, nor for any infringement of patents or other rights of third parties resulting from its use. No license is granted under any patents, trademarks or other rights of **NM**.

The author, **NM** Numerical Modelling GmbH, makes no representations, express or implied, with respect to this documentation or the software it describes, including without limitations, any implied warranties of merchantability or fitness for a particular purpose, all of which are expressly disclaimed. The author **NM**, its licensees, distributors and dealers shall in no event be liable for any indirect, incidental or consequential damages.

**NM** wishes to record its gratitude to the numerous individuals that contributed, either through discussions, proposals, support or testing, to the development of this project. In particular, we wish to thank Prof. Dr. Edoardo Anderheggen who has over many years supported the design and development of *SESES*. The following institutions supported the authors during the initial development of *SESES*: Swiss Federal Institute of Technology in Zürich, the Commission for Technology and Innovation of the Swiss Federal Government and Landis & Gyr Zug.

# Contents

<b>1</b>	<b>Introduction and the basics of SESES</b>	<b>5</b>
1.1	Installation . . . . .	7
1.2	Input Files . . . . .	8
1.3	Front End Program . . . . .	8
1.4	Kernel Program . . . . .	9
1.5	Output Files . . . . .	9
1.6	Embedding files in a single container file . . . . .	10
<b>2</b>	<b>Front End Program</b>	<b>12</b>
2.1	Main Toolbar . . . . .	12
2.2	Text Editor . . . . .	14
2.3	Mesh Builder . . . . .	15
2.4	Visualization Engine . . . . .	17
<b>3</b>	<b>Syntax Diagrams</b>	<b>24</b>
3.1	Numerical Values . . . . .	25
3.2	User Parameters . . . . .	31
3.3	Literals and Strings . . . . .	32
3.4	Classes of built-in parameters and functions . . . . .	33
3.5	Block Functions . . . . .	35
3.6	User Routines . . . . .	36
3.7	Text Preprocessor . . . . .	38
3.8	Input Units . . . . .	42
3.9	Input Syntax . . . . .	43
3.10	Syntax of the initial section . . . . .	44

3.11 Syntax of the command section . . . . .	62
<b>4 Numerical Models</b>	<b>89</b>
4.1 Finite Element Mesh . . . . .	92
4.2 Finite Elements for the Laplace equation . . . . .	95
4.3 Non-Linear Solution Algorithms . . . . .	98
4.4 Continuation Methods . . . . .	100
4.5 Unstationary Solutions . . . . .	105
<b>5 Physical Models</b>	<b>110</b>
5.1 Governing Equations . . . . .	110
5.2 Material laws . . . . .	128
5.3 Boundary Conditions . . . . .	142
5.4 Tensors and their Representation . . . . .	144

# Chapter 1

## Introduction and the basics of SESES

The program system *SESES* is a general purpose multiphysics numerical simulation tool for the analysis of micro-macro, sensor-actuator or generic devices by the finite element method on two- and three-dimensional domains. The program is especially useful during the design of systems, where modeling tools are used to optimize prototypes before they are manufactured. Other important uses of *SESES* are the investigation of manufacturing tolerances and the inverse modeling problem. Finally, *SESES* is excellently suited to the instruction of engineering students, since it supports the intuitive exploration of complex phenomena including their visualization. *SESES*'s area of applications ranges from the simple one-field model described by the linear Poisson equation to more complex non-linear and multiphysics models required for sensors and actuators. In general, it may be used to model coupled problems stemming from the linear or non-linear analysis of electrostatics, magnetostatics, current flow, eddy currents, elasticity, thermal flow, fluid flow, reactive mass flow and semiconductor drift-diffusion. Almost all of these equations can be coupled and solved together. A number of original numerical procedures are implemented in *SESES*, some of them are mentioned below.

- Fully preprocessed textual input combined with a functional input language for high flexibility data input and model setup. Just-in-time compilation of user functions provides a fast evaluation.
- Functional definition of geometries, material laws, solution algorithms and support to work directly in read-write mode with internal quantities like user allocated memory, dof-fields and solution updates.
- The finite element mesh is initially built from tensor-product blocks of cubic cells, each cell consisting of hexahedral, tetrahedral and prismatic macro elements for 3D and quadrilateral and triangular macro elements for 2D. Some of these macro elements may be deleted or distorted in any direction to correctly map the modeling domain and the contact geometries. Macro elements belonging to different blocks can be freely joined together by connecting faces.

- To increase the efficiency and accuracy of computations, each single (3D) hexahedral and (2D) quadrilateral macro element can be recursively refined in eight and four sub-elements. For these irregular generated meshes, special continuity conditions for hanging nodes are required between elements with a different refinement level. *SESES* guarantees these continuity conditions as well as the removal of equations associated with hanging nodes, leading to an overall reduction of equations to be solved. Adaptive meshes with hanging nodes for any macro element's shape are under construction.
- *SESES* supports adaptive a priori or a posteriori mesh refinement according to a user defined indicator. For example, after the computation of some dof-fields on a first coarse mesh, the mesh can be refined according to a posteriori error estimator leading to near optimal meshes for a large class of practical problems.
- For complex modeling problems with multiple fields and coupled effects, some of the fields may be negligible in some regions of the domain. The equation concept allows the implicit elimination of single fields on selected regions, thereby saving on computational resources.
- Sensor or actuator behavior is based on coupled effects. Various coupling mechanisms have been classified according to their type and have been implemented. For the solution of these multiphysics problems, either a generalized fully coupled Newton's algorithm or a sequential decoupled Gauss-Seidel algorithm is available.
- For postprocessing, besides predefined output fields, the user can define new fields as functional expressions of the available ones. Textual output can be freely formatted by the user in order to easily pass output data to other postprocessing programs.

*SESES* is a suite of programs written in the C-programming language and supported on most UNIX<sup>TM</sup> and Windows<sup>TM</sup> operating systems. User friendliness is provided by the Front End interactive GUI program allowing the definition and visualization of a modeling problem together with the analysis of numerical results produced by the computational batch-oriented Kernel program.

After describing the basics of *SESES*, this manual continues in Chapter 2 *Front End Program* by presenting the graphical tools and in Chapter 3 *Syntax Diagrams* by explaining the syntax of the input files. In Chapter 4 *Numerical Models*, we present some theory related to the numerical algorithms and the Chapter 5 *Physical Models* presents the available governing equations to be solved together with the associated material laws and the boundary conditions. A *SESES*'s beginner should finish to read this chapter together with an overview of Chapter 2 *Front End Program* to learn the functionality of the GUI. Afterwards, some examples installed in the template directories should be investigated, whereas it is recommended to modify them according to the own ideas. After the user has gained some experience, more complex simulation problems can be investigated right away.

## 1.1 Installation

The *SESES* software is available for Windows and Linux and can be download at the website `www.nmt.ec.ch` of the licenser, Numerical Modeling GmbH.

### Installation under Windows

The Windows version comes as a MSI package `NMSeses.msi` for the standard Windows installer. Once downloaded, click on the package to start the installation process.

### Installation under Linux

The Linux version comes as a tar-ball package `NMSeses.tgz` and installation is performed with the command line statement `tar zxvf NMSeses.tgz` in a directory of your choice. If you run *SESES* from the command line, update the `PATH` shell variable to include the *SESES* `bin` directory. Alternatively, you can define the associations with the file endings `.s2d`, `.s3d` and the executables `g2d`, `g3d` in the control center of your distribution in order to start the Front End program by selecting *SESES* files inside the browser.

### Getting a License

All examples included within this installation including all tutorial examples can be run without the need to acquire a *SESES* license. You may even run similar examples by changing and adapting to your needs the statements of the command section, e.g. to visualize numerical results or to produce data for post-processing tasks. However, a valid license is required to take full advantage of all features and to freely define statements of the initial section. To obtain a license, we need the host information obtained by running the programs `g2d` or `g3d` and by selecting the `Host Info` item from the `File` menu.

### Directory Structure

After a successful software installation, the target directory contains the user manual `Manual.pdf`, the tutorial `Tutorial.pdf` and the following subdirectories:

- **bin:** The executable are contained in this directory.
- **example:** This subdirectory contains several examples, including the ones discussed in the tutorial. The user may browse in these example files and explore many *SESES* features.
- **include:** The include files with routine definitions are located here.
- **test:** Several input files are included in this directory for testing purposes.

## 1.2 Input Files

An input text file with default name `Seses` is used to specify a modeling problem. The file is read both by the Front End and the computational Kernel program and consists of an initial and a command section. The initial section is used to describe the modeling problem, as for example, the domain geometry, the boundary conditions and the physical models. The command section is used to specify the numerical algorithms, as for example, the type of linear solver, the non-linear solution algorithm and the generation of output data. The main reason for defining two separate sections is that the initial section defines static data easily visualized within the Front End program, whereas the command section defines dynamic data just available during the simulation. The required syntax for these files and to a large extent their semantics, i.e. the meaning of the data they contain, are described by syntax diagrams. As visual aids, we use here syntax diagrams in the form of railway lines with stations and switches. The syntax results by traversing a diagram as a train does a railway line. These diagrams and their semantic are presented and explained in detail in the Chapter [3 Syntax Diagrams](#).

To customize the appearance and operation of the Front End graphical program, on a UNIX™ system configuration options are automatically stored in the `.sesesrc` file in the user's home directory and on a Windows™ system in the registry under the entry `HKEY_CURRENT_USER/Software/Numerical Modeling GmbH`. For example, we store here editor and printing options defined by the user. However, options used to create graphical pictures are stored in a separate `SesesSetUp` text file, since they are considered problem's dependent. This file is generated on request within the Front End program and if defined, the file is read together with the input file.

## 1.3 Front End Program

The Front End program presented in the Chapter [2 Front End Program](#) is an interactive GUI program for the definition and visualization of simulation domains, finite element meshes and numerical results. The visualization engine is based upon the OpenGL APIs and OpenGL drivers should be installed for a fast graphical representation. Actually two Front End programs are available `g2d` and `g3d`, the former for 2D and the latter for 3D modeling domains.

The Front End program is started by typing in a command shell, the program name `g2d` or `g3d` followed by an optional file name. If the file name is not given, the default name `Seses` is used. It is customary to use the file ending `.s2d` or `.s3d` for *SESES* files belonging to the 2D or 3D version, so that if the correct file associations have been defined during installation, the most common way to start the Front End program with a given input file, is given by clicking on the file inside a browser. Upon invocation of the Front End program and if the input file is syntactically correct, the modeling domain is initially visualized, otherwise a text editor displays the input file and highlights the first line with an error. Here, with the help of the error messages, the user should be able to correct the error.



The following options are recognized by the Front End program:

-p	Write a print file and terminate.
-po <file>	Specify the print file base name, default is Graph.
-r <file>	Specify the graphics setup file name, default is SesesSetUp.

## 1.4 Kernel Program

The computational Kernel program is a batch-oriented finite element program. Actually two Kernel programs are available k2d and k3d, the former for 2D and the latter for 3D modeling domains. In general, they are quite parsimonious in the usage of computer resources so that many 2D and 3D problems of interest can already be run on desktop computers and only medium to large 3D simulations require larger computational resources. The Kernel program requires an input file to describe the modeling problem whose syntax is thoroughly discussed in the Chapter [3 Syntax Diagrams](#).

The Kernel program is started by typing in a command shell the program name k2d or k3d followed by an optional file name. If a file name is not given, the default name Seses is used. Another common and more simple way to start the Kernel program is with the help of a button inside the graphical Front End program. The Kernel writes error messages, some few default informations and user messages to the standard output stream. If syntax errors are detected in the input file, the Kernel program stops its operation. Here, with the help of the error messages, the user should correct the error and start the Kernel program again. It is however more practical to correct the error inside the Front End program, instead of using a common text editor and the error line number.

The following options are recognized by the Kernel program:

-h	Write host license informations and terminate.
-u	Give a list of defined but unused routines and terminate.
-w	Wait for an input from the keyboard before terminating.
-e	Wait for an input from keyboard before terminating but just when errors occur.
-z	Terminate on soft errors.
-s	Just check the input files and exit.
-n	Generate mathematical library errors (default on Windows).
-t <num>	Run in parallel mode with <num> threads (default is one).
-i <txt>	Define the text <txt> as starting input text.

## 1.5 Output Files

To visualize numerical results, the Kernel program has first to write graphics data files which are then read by the Front End program. A graphics data file generally contains the information on the finite element mesh and one or several fields of interest. An

ASCII or binary format is available, the former ensures portability among different computer platforms and the latter a compact format.

To restart a computational session from a given point, state data files can be written by the Kernel program describing an instantaneous state of a solution algorithm on a finite element mesh. These files may only be read by the Kernel program, not by the Front End program. For state data files is recommended to use the binary format which represents a true copy of the memory content. With the ASCII format instead, the input-output conversion of numerical values between the binary and decimal notation may lead to small differences in the data.

Upon request from the user, the Kernel program generates output ASCII files to supply all relevant numerical data. Most textual output can be freely formatted in order to easily pass the data to other postprocessing programs like plotting routines and working sheets. Just in some rare occasions, a text filter must be placed in between the two applications. It is for example possible to define a lattice of points in the space and to supply the numerical values of any output fields at these points or to integrate these fields over any part of the domain or its boundary.

## 1.6 Embedding files in a single container file

This section presents a somewhat hidden but very practical feature of *SESES*. As briefly explained in the Section [1.2 Input Files](#), to run a modeling problem an input file must be defined and optionally a second *SesesSetUp* file for the graphical setup, but other files may be as well requested by the input. If in a directory several modeling problems are defined, it may be inconvenient to manage all these files and the preferred way would be to have a single input file for each different modeling problem.

A practical solution is offered by the possibility to embed all required input files in a single container file. In this case, inside the container file, the files must be defined in one of the following two ways

```
cat > <filename> << <EndToken>
...
<EndToken>

cat >! <filename> << <EndToken>
....
<EndToken>
```

The content of the file *<filename>* starts at the next line after the first *<EndToken>* string and ends with a newline character at the line before the second *<EndToken>* string. The *<EndToken>* string may be any string without space characters.



When the *SESES* programs start reading the input file, they first check if a container file has been defined by trying to locate the embedded *Seses* file inside the container file. Container files may also contain the *SesesSetUp* file as well as other files used by the input parser. A container file may also be a script file that can be executed, as presented in the next example, for the case of a Bourne shell on a Unix operating system.

```
#!/bin/sh
# Create the Seses file
cat > Seses <<EOF
QMEI
...
Finish
Solve Stationary
Dump Psi
...
Finish
EOF
k2d # Run the computational Kernel
```

If the above shell script has execute permissions, by typing its name in a command shell, the *Seses* file is created and the *k2d* Kernel program is executed. The same simulation is also started if one passes the name of the script file to the program *k2d*. In this latter case no *Seses* file will be created and the execution is a little bit faster since the script file is not interpreted by the shell. Running the Kernel programs by running the script files, however, offers the possibility to add other commands to be executed before or after running the Kernel program.

## Chapter 2

# Front End Program

The Front End program is an interactive GUI program allowing the user to conduct a complete *SESES* modeling session without leaving the application, thus starting from the definition of a modeling problem up to running the simulation in the background and viewing the numerical results. In particular, the 2D version has a mesh builder allowing the graphical construction of the mesh as alternative to a textual and functional definition or mesh importation as done for 3D. To visualize a modeling problem only the initial section of the input file containing all geometrical informations is required whereas to display numerical results, data files generated by the computational Kernel program are needed. Upon invocation of the Front End program, a new window is displayed on the computer screen with a horizontal main toolbar at the top of the window. If the input file is syntactically correct, something like in Figs. 2.1 will appear. Below the main toolbar there is a graphics window where the modeling domain is displayed. On the right side several control panels are available to define the contents of the graphics window. If syntax errors are detected, instead of the graphics window, a text editor with the input file is displayed and the syntax error is highlighted. Together with the error message displayed in the output window placed below the editor window and delimited by a horizontal bar, the user should be able to correct the error. By clicking on the graphics  button, the file is saved and newly parsed. By default, the editor window occupies the full size and hides the output window, but it is possible to resize both windows by grabbing the mouse pointer over the horizontal bar or by using the  button on the main toolbar. The output from the Kernel program is written to a separate console window.

### 2.1 Main Toolbar

The main toolbar is the horizontal control bar placed below the top window's bar containing the application's name, see Fig. 2.1.

The **FILE** button is a pull down with the options **NEW** to define a new empty file, **OPEN** to load a file, **SAVE** to save the file, **SAVE AS** to save the file into another file, **KILL** to kill a running batch job, **HOST INFO** to print host info for license purposes, **SETTING** to set up application's options and **QUIT** to exit the Front End program.

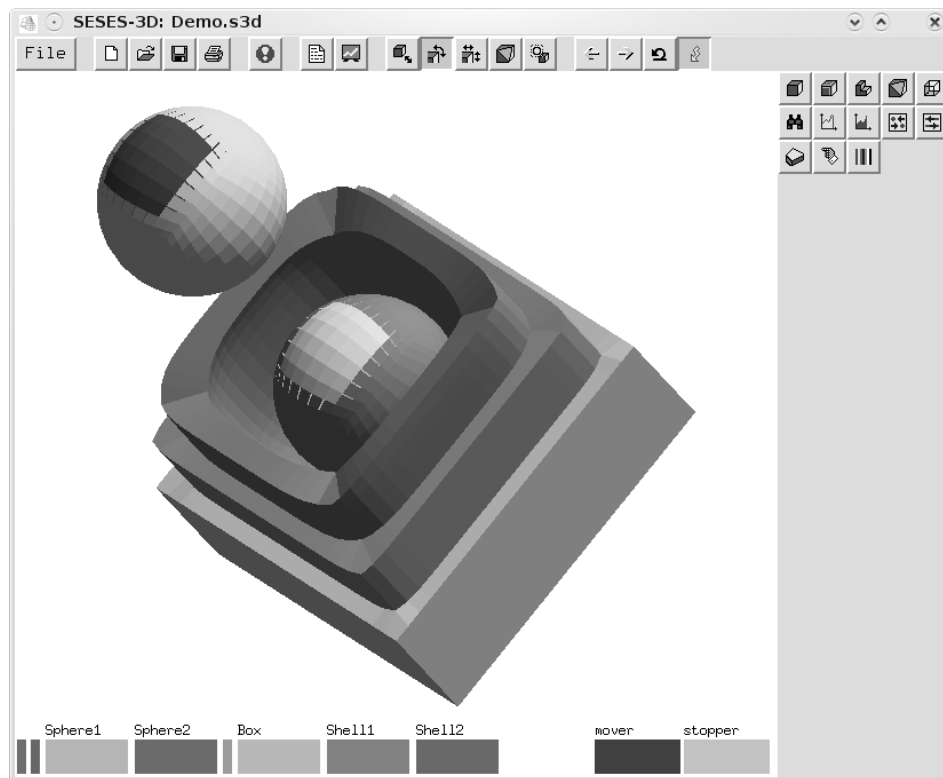














Figure 2.1: The Front End program.

This pull down menu also contains several names of previously loaded files for quick reloading.

The open file  button and the save file  button are equivalent to the **OPEN**, **SAVE**, options of the **FILE** pull down. The export image  button writes a file with the content of the graphics window. Per default the file is a Portable Network Graphics file with an index starting from zero (Graph000 .png). If the target file already exists, the index will be incremented. JPEG and EPS encoding is also supported as well as other useful options, selectable with the image setup panel **FILE** → **SETTING** → **IMAGE**.







The graphics  button reads and parses the input file and if free of syntax errors the visualization engine is started to display the structure, see the Section 2.4 *Visualization Engine*. The input file will be saved automatically if any changes are made. The  button starts the text editor for editing the input file, see the Section 2.2 *Text Editor*. For the 2D version, the  button starts the mesh builder for constructing the computational mesh graphically, see the Section 2.3 *Mesh Builder*. Within the text editor, mesh builder or visualization engine, the following shortcut key events are available:


Operation	Key Event
Save the file	Control-s
Quit the application	Control-q
Save the file and start the Kernel	Control-r
Save the file and start the text editor	Control-e
Save the file and start the mesh builder	Control-b
Save the file and start the visualization engine	Control-g

The  button starts the Kernel program in the background. Per default, the Kernel program is called with the file name of the current loaded file as parameter, but it is possible to define other commands with the run setup panel  →  → . The  dialog determines if the commands to be executed should be run directly or passed as parameters to a shell program. In the former case when no shell is defined, the first string of a command must be a runnable program and subsequent strings are passed as parameters. If a shell is defined, the first string defines the shell program, subsequent strings are passed as parameters to the shell and the command to be executed is passed as last and single parameter. Thereafter, three different commands to be executed can be defined selectable with a radio button set. To help constructing a running command, the command is parsed for the placeholder symbols %f, %K and %B which are replaced by the filename of the actual loaded file, the name of the Kernel program and the directory of the *SESES* binaries. Finally, the  option determines if the Front End should parse the command section of the input file, otherwise just the initial section is parsed.

The main toolbar is enriched with additional buttons depending if either the text editor, the mesh builder or the visualization engine is active and they will be presented in the next sections.


## 2.2 Text Editor

A text editor displaying the input file is started with the  button. This enhanced editor supports syntax highlighting and can perform unlimited undo&redo operations which however are lost when another file is loaded. The text insert position, visualized by a small triangle, is changed by pressing the first mouse button at the new position. Before releasing the mouse button, text is selected (highlighted) following the mouse movements. A selection can be continued by holding down the `Shift` key, while pressing the first mouse button. By a fast double or triple click of the mouse button, the selection is limited to entire words or lines. Text can also be selected by using the arrow keys together with the `Shift` key. The editor works with a single buffer and the available operations are listed in Table 2.1 together with the corresponding keyboard events. Selected text can be placed in the buffer with a copy operation . With a cut operation , selected text is removed and placed in the buffer. A paste operation  inserts the buffer at the current insert position, while the buffer is left unchanged. With a find operation selected text, or if no selection is available the text stored in the buffer, is searched in the forward direction starting at the insert position. If the `Shift` key is also pressed the search is done in the backward direction. With a replace-find operation selected text is replaced with the text stored in the buffer and a new find operation is performed. With an undo operation  (backward undo) any previous action inside the editor is cleared. This may be repeated until reaching the state when the file was loaded. With the redo operation  (forward undo) any previous undo operation is cleared. This may be repeated until reaching the state when the undo operations were started.

If the editor was automatically started because syntax errors were detected within the input file, the  button is enabled and the line of text with the offending error is high-






Operation	Key Event	Operation	Key Event
Cut	Control-x	Page Up	Pg Up
Copy	Control-c	Page Down	Pg Dn
Paste	Control-v	Top	Control-h
Find Forward	Control-f	Bottom	Control-l
Find Backward	Shift+Control-f	Line Start	End, Control-j
Replace/Find Forward	Control-a	Line End	Home, Control-k
Replace/Find Backward	Shift+Control-a	Line Up	Arrow ↑
Undo Backward	Control-z	Line Down	Arrow ↓
Undo Forward	Shift+Control-z	Previous Char	Arrow ←
		Next Char	Arrow →

Table 2.1: Keyboard events for the text editor.

lighted. However, if the syntax error was caused inside an included file, just the file inclusion will be highlighted and by pressing the  button, the user can recursively step into the hierarchy of included files until reaching the file with the offending error.







The font used by the editor, its size, text color highlight and other configuration parameters are defined with the editor setup panel `FILE` → `SETTING` → `EDITOR`. In particular, it is possible to define commands and bind them to the keyboard F-keys or to the editor pull down menu activated with the right mouse button. If the command is defined as a filter, the selected text in the editor is passed as input to the command and it will be replaced by the command's output. The calling convention for these commands is the same as for the Kernel programs explained on p. 14. However, when forming the command we have the additional placeholders %s, %b and %S representing the filename of a temporary file containing the selected text, the filename of a file containing the text in the buffer and the selected text.

## 2.3 Mesh Builder


The graphical mesh builder is only available for the 2D version and it is started with the  button. The builder works like an editor with selections, copy&paste operations, mesh related operations and unlimited undo&redo operations. The mesh builder just reads a special section of the input file placed in-between the keywords MeshBuilderStart and MeshBuilderEnd to obtain the mesh data and whenever you leave the builder, only this section will be updated. The builder understands only a very restricted subset of the functional input syntax and therefore this section should not be edited manually. Within the mesh builder, you can easily define, update and delete MEs, materials, domains, boundaries, joins and curves. Within the main toolbar, you can use the button  to set the raster&snap properties when constructing the mesh and to display a background image. The buttons , ,  determine the behavior of the mouse wheel, either zooming in-out with respect to the mouse pointer position, shift left-right or shift up-down. Wheel's operations are permitted also when dragging objects and by pressing the wheel button, you can quickly change its behavior. You can use the option `IMPORT` of the `FILE` pull down menu, to import a previously constructed mesh and add it to the actual one. To convert a valid SESES mesh













into the builder format, you can use the `Write MeshSetting` statement of the 2D Kernel program.

Single objects are selected by clicking on them with the first mouse button. If nothing get selected or if the `Ctrl` key is hold down, you can move the pointer to select all objects within the highlighted rectangular region or also within a polygonal region if the button  is selected. A selection can be continued by holding down the `Shift` key. The builder works similarly to the editor with a single buffer. Selected objects can be placed in the buffer with a copy operation . With a cut operation  selected objects are deleted and placed in the buffer. A paste operation  inserts the buffer at the current mouse pointer position, while the buffer is left unchanged. With an undo operation  (backward undo) any previous action inside the builder is cleared. This may be repeated until reaching the state when the builder was started. With the redo operation  (forward undo) any previous undo operation is cleared. This may be repeated until reaching the state when the undo operations were started. You can also use the undo&redo features of the editor, to undo&redo complete working sessions within the builder.

The single builder operations are activated with the toolbar on the right. Within each option, the selection&cut&copy&paste operations have in general a different behavior adapted to the current task.







The button  inserts or geometrically modifies rectangular ME quads. The insert operation is started and ended by a double click at the first and last quad nodes, the other quad nodes are defined in-between by a single click of the first mouse button. In this operation mode, selected MEs, edges or nodes can be moved. Use a cut operation on selected MEs to remove MEs.








The button  inserts and modifies Bézier or NURBS curves. The insert operation is started and ended by a double click at the first and last control point, the other control points are defined in-between by a single click of the first mouse button. To add a control point to a curve by leaving the curve shape invariant, double click at the insert position on the curve. Use a cut operation to remove selected control points. Because curves are generally constructed using a background image, they are adapted for interpolation and do not work with non-interpolating control points. Instead curves consist of segments defined by two interpolating control points and two tangent vectors determining either a polynomial (Bézier) or rational (NURBS) line segment. In this operation mode, selected control points are changed with the help of the displayed dumbbell. Symmetric, parallel and discontinuous tangent transitions are set with the buttons ,  and . The curve type of a segment is changed by selecting its two ending points and the button  or . To close a curve or to connect it to another one, check the button  and drag the first control point over second one until a snap takes place. To open or cut a curve in two parts use the button  on selected control points. Use the button  to open a panel for defining a circular arc between two selected control points.

The button  attaches nodes of the mesh to a curve or modifies them. Selected mesh nodes are attached to a curve by dragging the selection over the curve until a snap takes place. During the snap, the mesh nodes are distributed along the curve by moving the pointer along the curve. Therefore, by selecting singularly attached nodes can





move them along the curve. By additionally pressing the `Shift` key, the initial snap point is moved along the curve and with the `Control` key, you can force releasing the actual snap and a new one can be started.




With the button  and by checking either one of the buttons , , , or , you can move, rotate and even or unevenly resize selected objects by dragging the selection. Use the button  to open a panel for defining the affine map numerically.

With the button  and by checking either one of the buttons ,  or  you join together ME edges by dragging selected edges to the new location until a snap takes place. With  you can only join free nodes not being attached to curves, with  any nodes and with  you can only join unconnected MEs but all connected MEs and curves are automatically resized to best fit the joined edges.



With the button , MEs are split along selected edges. Use a cut operation on selected edges for the reverse operation.

With the button  you can move the split which by default lies in the middle of the edge.


With the button , you can smooth the internal nodes of the selected MEs. A fast global interpolation method of the selected boundary nodes is available which in general does not work well if the geometry is non-convex.





With the buttons , , , materials, domains and boundaries are mapped on selected mesh items. Use a cut operation on selected items for the reverse operation. You can change the active material, domain or boundary either with these buttons and the third mouse button displaying a pull-down menu of available choices or with the help of the mouse wheel by contemporary pressing the `Shift` or `Control` key. A double click in the builder window displays a panel to define and manage materials, domains or boundaries.



## 2.4 Visualization Engine

The visualization engine is started with the  button. Below the main toolbar, we have a graphics window used to display the modeling domain together with numerical results. On the right side, we have several control panels used to define the contents of the graphics window which are opened by selecting the corresponding radio button, e.g. the  button for the material control panel.

Other configuration parameters not changed so often are defined with the graphics setup panel `FILE` → `SETTING` → `GRAPHICS`. The `LENGTH FORMAT` dialog specifies the format and length unit used to print length values inside the graphics window when clicking the middle button on a displayed structure. The format is defined first and must conform to the ANSI C standard for the `printf` routines and the `float` type, then a hyphen and the length unit must follow. The `VALUE FORMAT` dialog specifies the format used to print field values inside the graphics window when clicking the middle button on a displayed field. The format must conform to the ANSI C standard for the `printf` routines and the `float` type and the additional escape sequences `%F`, `%C` and `%U` are available to print the field name, the component name and the unit name.











The reset  button has two modes of operation. With a single click, the picture viewport is newly computed in order to render fully visible the structure inside the graphics window. Although this operation is always performed after reading a new input file, the structure may not fit anymore inside the window after a zoom, a view-angle or a move selection. With a fast double click, relevant graphics settings are returned to their default values. How to define default values is explained on p. 22.

The ,  buttons are used to change the default graphics state by selecting the previous or next graphics state as new default state. How to define default values is explained on p. 22. This possibility is provided for quickly viewing different pictures by just clicking on the single ,  buttons, instead of setting each time several graphics parameters.

Since a new picture may require several graphics settings to be set, the graphics window is only redrawn by a change of these settings, only if the update  toggle is selected. If deselected, the changes are only visible by a manual redraw with the graphics  button.


## Graphics window

The graphics window has some hidden features with respect to mouse events, therefore for a beginner it is important to read this section.

The first mouse button is used for selection. In 3D, five different selections are available activated by the toolbar buttons , , ,  and , the last two ones are not defined in 2D. By selecting the  button you can change the zoom. Just press the first mouse button, hold down the button and drag the mouse in any direction. The zoom window boundary will be highlighted. By releasing the mouse button you end the zoom selection. By dragging the mouse on the left side or above the left-upper corner of the zoom window boundary you can move this corner even outside the graphics window. If the zoom window boundary lies completely outside the graphics window, instead of a zoom in you then obtain a zoom out effect. By selecting either the  or  button, you can change the view direction or the view position. Press the first button and drag the mouse, horizontal and vertical mouse movements will then correspond to rotations around the vertical and horizontal axes or to horizontal and vertical translations. At the same time, you can quickly toggle between rotations and translations by pressing the third mouse button. By selecting the  button, you can define or change a 3D-cut. Press down the first mouse button on a point of the structure, where you want the 3D-cut to go through. It will be defined corresponding to the cut direction set in the view control panel. By dragging the mouse, horizontal and vertical movements will correspond to rotations of the 3D-cut around the vertical and horizontal axes and with the mouse wheel you can shift the 3D-cut. If when pressing the button you do not select any point of the structure, the 3D-cut is disabled. By selecting the  button, you can change the light-source position. Press the first button and drag the mouse to change the in-coming direction of light.

By clicking with the second mouse button inside the graphics window on some object, a numerical value is printed at the mouse pointer location. The content of the value depends on the type of object selected. If the modeling domain is displayed, the world

coordinates of the selected point are printed, if a field is displayed the numerical field value is printed instead. For vector fields, this value may be the absolute vector value or a single coordinate value.

By clicking the third mouse button, the graphics window is redrawn. This action is the same as activating the graphics  button.

The mouse wheel can be used to zoom in and out. While scrolling, the mouse pointer defines the position of the object that will stay fixed on the screen. Generally, this is a more convenient form than dragging the mouse as described above.

If a field is currently displayed, on the right side of the graphics window a legend of field values is displayed. If you enter with the mouse pointer the legend, the color you have selected will be highlighted inside the legend and inside the graphics window. This feature allows the user to easily visualize isoline contours for a displayed field.

If the modeling domain is displayed, at the bottom of the window a material and a BC table are displayed defining the colors used for each items, see Fig. 2.1. By clicking with the first mouse button on a table entry, you can deselect an item from being displayed or select it again.

The content of the graphics window can be stored into the clipboard by typing the `Control-p` key event.

### Material control panel

With this panel, settings controlling the drawing of materials are defined. The choice menu lists all materials and by enabling/disabling the materials with a double click, just MEs belonging to enabled materials will be drawn. For the selected material in the choice menu, by disabling the `APPLY MAP` toggle, we do not display numerical fields on this material and when transparency is used, the `TRANSPARENCY` dialog allows to specify an additional material transparency factor. With the `ME GRID` toggle enabled, the ME grid is drawn when displaying the modeling domain. The `MAT. TABLE` toggle turns on/off the material table at the bottom of the graphics window permitting to quickly enable/disable materials by clicking with the first mouse button on the table entries, see Fig. 2.1.

### BC control panel

With this panel, settings controlling the drawing of BCs are defined. It works as the material control panel but has less options.

### Domain control panel

With this panel, settings controlling the drawing of MEs are defined. These settings underlie the material control settings and MEs belonging to disabled materials cannot be enabled here. The choice menu lists all domains and by enabling/disabling the domains with a double click and by combining them with one of the logical operations,

it is possible to tailor the drawing to single MEs.

### Cut control panel

The **CUT** toggle enables/disables a cut selection. The disable action is equivalent to pressing the first mouse button inside the graphics window on immaterial points when the structure-cut selection is active. The **FLIP CUT** button allows to switch between displaying either the domain on the left or the domain on the right side of the cut. The **SET CUT DIR** pull down allows to set a predefined cut direction. The **SHIFT** dialog defines the distance of the cutting plane from a point approximately in the middle of the modeling domain in the direction normal to the plane.

### Transparency control panel

With this panel, settings controlling the transparency of the picture are defined. Transparency is used in order to have an in-depth view in 3D and can be singularly enabled for materials, maps or dots. When transparency is enabled, you should define a transparency factor less than one, otherwise solid colors will be drawn hiding everything behind. You can define an additional transparency factor for each single material with the material control panel.

### View control panel

With this panel, settings controlling the view of the modeling domain are defined. In particular, the panel defines additional settings not available with the selection mechanisms of the graphics window for the zoom, view-angle, viewport-move and the 3D structure-cut selection, as described in the Section [2.4 Visualization Engine](#). The **SET VIEW POINT** pull down allows the selection of six predefined view-angles and the **ZOOM** pull down allows to zoom in/out or to a predefined factor. The **RASTER** dialog defines the raster for the view and cut angles. For a value different from zero, the angles expressed in degree will be set to a multiple of the raster value when a view or cut angle is changed, see Section [2.4 Visualization Engine](#).





### Parameter control panel

With this panel, material parameters just depending on known input values are displayed. Material parameters not requiring data files for their representation are listed in the choice menu and a selected parameter is draw by enabling the **DRAW** toggle. The **HOMO RL** dialog defines a virtual mesh obtained by refining homogeneously each ME the specified number of times and used to evaluate and draw the material parameter. The other options are the same as for the field control panel where they are discussed.

## Field control panel

With this panel, numerical fields defined by data files are displayed. Data files are generated by the Kernel program with the Dump statement and several data slots can be added to the same file with the option Append as described on p. 83. The numerical fields defined by the numerical data are listed in the **FIELD** choice menu and a selected field is draw by enabling the **DRAW** toggle. When displaying a field with a vector character, the **COMP** pull-down allows the representation of a scalar value associated with the non-scalar field. For example, if the field is a vector field, the norm or each single component is available for selection.

The name of the data file is given by the **FILE** dialog and if the data file defines several slots, the **SLOT** dialog allows the slot selection. The insert position of the **FILE** dialog can only be changed with the middle mouse button. The left and right mouse buttons are instead reserved to quickly change the numerical suffix of the file name without editing the **FILE** dialog. By pressing the right mouse button, the numerical suffix of the file name is incremented by one, if no suffix is given a zero will be considered as the first suffix. With the left mouse button, the numerical suffix is decremented by one until the zero value is reached; by further pressing the left button the numerical suffix is removed.

The **ANIMATION** pull down allows to consecutively display the same field in each data slot; each new picture is displayed as quickly as possible in order to create an animation. The options , , ,  determine the order in which the slot data is traversed in one single cycle. With the first and second option, the field data is displayed in increasing or decreasing order of the slot number. The second and third options are a combination of the first two options. In one cycle the field data is displayed first in increasing then in decreasing order of the slot number, respectively first in decreasing and then in increasing order of the slot number. The selection of one of the options **ONE CYCLE**, **TWO CYCLE** or **GO** starts the process of displaying one, two or infinitely many cycles. This process is stopped by clicking on any mouse button inside the graphics window. The option **SAVE ONE CYCLE** creates the same animation as **ONE CYCLE** but will save each picture to disk.

If displacement fields are defined inside the data file, their names will appear inside the **ADD DISP** pull down and if the corresponding name is selected, the value of the displacement field is added to the geometric coordinates. The **DISP ANIMATION** option starts a periodic sine animation which is stopped by clicking on any mouse button inside the graphics window. If the **SAVE ANIMATION** option is selected, the animation will stop after one period and all the pictures are saved to disk. The **DISP FAC** dialog sets the amplification factor for the added displacement.

The **LEGEND** toggle turns on/off the legend of field values inside the graphics window and the **GRID** toggle enables/disables the drawing of the numerical grid in addition to the numerical field. The **CONTOUR** pull down defines the mapping style. With the **CONTOUR FILL** option, solid colors are used to fill field levels, with the **CONTOUR LINE** option, contour level lines are drawn and with the **MATERIAL** option, the material structure is draw as first and eventually will be completely hidden, if solid field levels are drawn without transparency. The number and distance of the contour levels is set with the miscellaneous control panel.

When the **MIN MAX** toggle is enabled, the minimum and maximum field values are determined by the **MIN** and **MAX** dialogs. Smaller and larger field values will be displayed with the colors of the minimum and maximum values. If the **MIN MAX** toggle is disabled, the minimum and maximum values for the displayed field are automatically updated in the dialogs.

### Dot control panel

With this panel, dots, arrows and triads are additionally drawn to help the visualization of scalar, vector and tensor fields. With the **ON ELEMENT** toggle enabled, the dots are drawn in the center of each element; it is a quick and fast method but not always optimal. Therefore, all lattices are listed in the choice menu and by enabling/disabling the lattices with a double click, a dot is drawn at each lattice point of the enabled lattices. By enabling the **FIELD COLOR** and the **FIELD SCALE** toggle, the dots have the same color of the contour levels and are scaled with respect to the maximal absolute field value. With the **DOTS** toggle enabled, dots are always drawn instead of arrows for a vector field and triads for a symmetric tensor field. The **DIMENSION** dialog defines a global scaling factor for the dots.


### Streamline control panel

With this panel, streamlines are additionally drawn to help the visualization of vector fields. All lattices are listed in the choice menu and by enabling/disabling the lattices with a double click, a vector streamline is displayed starting at each lattice point of the enabled lattices. By enabling the **FIELD COLOR** toggle, the streamlines have the same color of the contour levels. With the **TUBULAR** toggle enabled, thick tubes are drawn instead of thin lines. The **DIMENSION** dialog defines a global scaling factor for the tubes. With the **FAST** toggle enabled, the streamline is drawn a little bit faster, but more memory is required here. The **INTEGRATION** radio buttons select the order of the Runge-Kutta integration algorithm and the **SUBDIV** dialog, the number of integration steps in each element.

### State control panel

With this panel state settings customizing the operations of the Front End are defined.

By **WRITE STATE** button creates a `SesesSetUp` file defining the actual setting of the visualization engine and all user defined graphics states. If the input file is a container file with an embedded `SesesSetUp` file, this embedded file will be overwritten, otherwise the file is created or will replace the one in the actual working directory. Thereafter when starting the `g3d` or `g2d` graphical programs or when opening a new file, the graphics states will be installed according to the content of the `SesesSetUp` file.

The **STORE STATE** button stores the actual setting of the graphics visualization engine as default graphics state. Afterwards, this default state can be restored by a double click selection of the reset  button.



The **NEW STATE** button is similar to the **STORE STATE** button, but before storing the settings, a new graphics state is created and defined as the new default state. The default graphics state can be changed with the **←** and **→** buttons.

The **DELETE STATE** button deletes the default graphics state and sets the next entry in the list of states as the new default state. The actual graphics settings are left unchanged.

The **PRISTINE STATE** button installs a pristine graphics state and it is used to super-seed any setting of the `SesesSetUp` file.

## Miscellaneous control panel

With this panel miscellaneous settings are defined.

The **LIGHTING** toggle allows the user to create light effects. The light source can be positioned with the light-angle selection, see Section [2.4 Visualization Engine](#).

The **JOINED FACES** toggle helps the visualization of joined MEs by displaying the internal ME joined faces when transparency is enabled.

The **SPECTRUM** pull down selects the color spectrum used when displaying a numerical field, i.e. the variation of the color with the numerical field values. For the **MIN. BLACK**, **MAX. BLACK**, **ZERO BLACK** selections, the black color will be the minimal, maximal or zero numerical field value. The **NO BLACK** selection uses a spectrum without the black color and the **BLACK-WHITE** selection a spectrum from black to white, better suited for grayscale printing.

The **TICS** dialog specifies the value's format and the number of tics when displaying the legend of field values. The format conforming to the ANSI C standard for the `printf` routines and the `float` type is defined first and it is followed by a `#` character and the number of tics.

The **CONT** dialog specifies the contour levels drawn when selecting the contour option. The syntax of the string has the form `v0#n0:v1#n1:...` with `v<>` a contour value and `n<>` the number of contours equally spaced up to the next given contour value. If the first or last contour value is missing, i.e. the string starts or ends with a number of contour levels `#n<>`, the value will be substituted with the minimal or maximal field value. The contour values must be defined in ascending order.

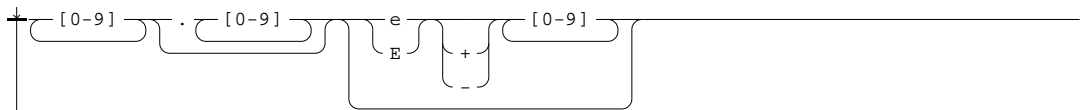
The **TILE** dialog specifies a title to be additionally displayed.

The color of each material or boundary as well as the color of the lines, dots, arrows and tubes can be customized. All these items are listed in the choice menu and with a double-click on a selected item, the color can be changed.

## Chapter 3

# Syntax Diagrams

*SESES* uses syntax diagrams in the form of railway lines, stations and switches to describe the syntax for the input files. The syntax diagrams are visual aids for the specification of the correct syntax and to a certain degree the semantics of the input language. Each diagram should be traversed along its lines as a train does a railway line.



All such routes are admissible within the specified syntax. Upon reaching a station, indicated by a thick horizontal bar, the syntax follows a horizontal line up to a terminal triangle. Repeated items are indicated by loops and optional items have by-passes. In order to satisfy the demand for high flexibility in the problem specification, a macro-language combined with a functional approach is available and some features of the input syntax are the following.

- Text formatting is free since all multiple return characters, tab stops and white-spaces are treated as a single word separator. Comments can be inserted anywhere in the input text. They start with ( \* and end with \* ) and may be nested.
- Preprocessing macros with parameters can be defined with the `Macro` statement, files can be included using the `Include` statement and input text can be generated with `For`-loops or `If-Else` statements.
- Whenever an integer or a real number must be given, an algebraic expression can be used to specify the numerical value, e.g.  $(\exp(3)+3)/4$ . For multiple input values, algebraic vector expressions are supported as well, e.g.  $(1, 2, 4)*3 + (3, 4, 5)/2$  and the complex algebra is available for vectors of dimension two e.g.  $3*(1, 2)/(3, 4)$ .
- User routines can be defined and used within any algebraic expression. The routine's code is based on common control statements and algebraic expressions. As an alternative, routines can be loaded from external libraries.



- To specify time dependent input entities, the user should use the built-in global variable `time` to describe the functional dependency. The `time` variable is just an example of a dynamic dependency, but any other user defined global variable can be employed and changed at run time.
- Many physical laws can be freely defined as functional expressions of some built-in symbols. For example, functions of the real coordinates  $x, y, z$  can be defined as functions of the built-in variables `x`, `y`, `z` or temperature dependent material laws as functions of the built-in variable `Temp`.
- When a physical unit is required to specify a numerical value, the user can give any unit consistent with the one required.

In the syntax diagrams, items starting with a capital letter are tokens that belong to the input language (*SESES* keywords), items starting with a lower case letter must be specified by the user and items in **bold** are allowed in the 3D-version only. In the case the `<>`-brackets surround an item, e.g. `<item>`, the item must be selected from a list. The syntax diagram does not directly report the list's content, but the user may view the full list just by giving a wrong item. In this case the input parser writes an error message the full list to the output stream, helping the user to quickly select the right item without the need to look at this user manual.

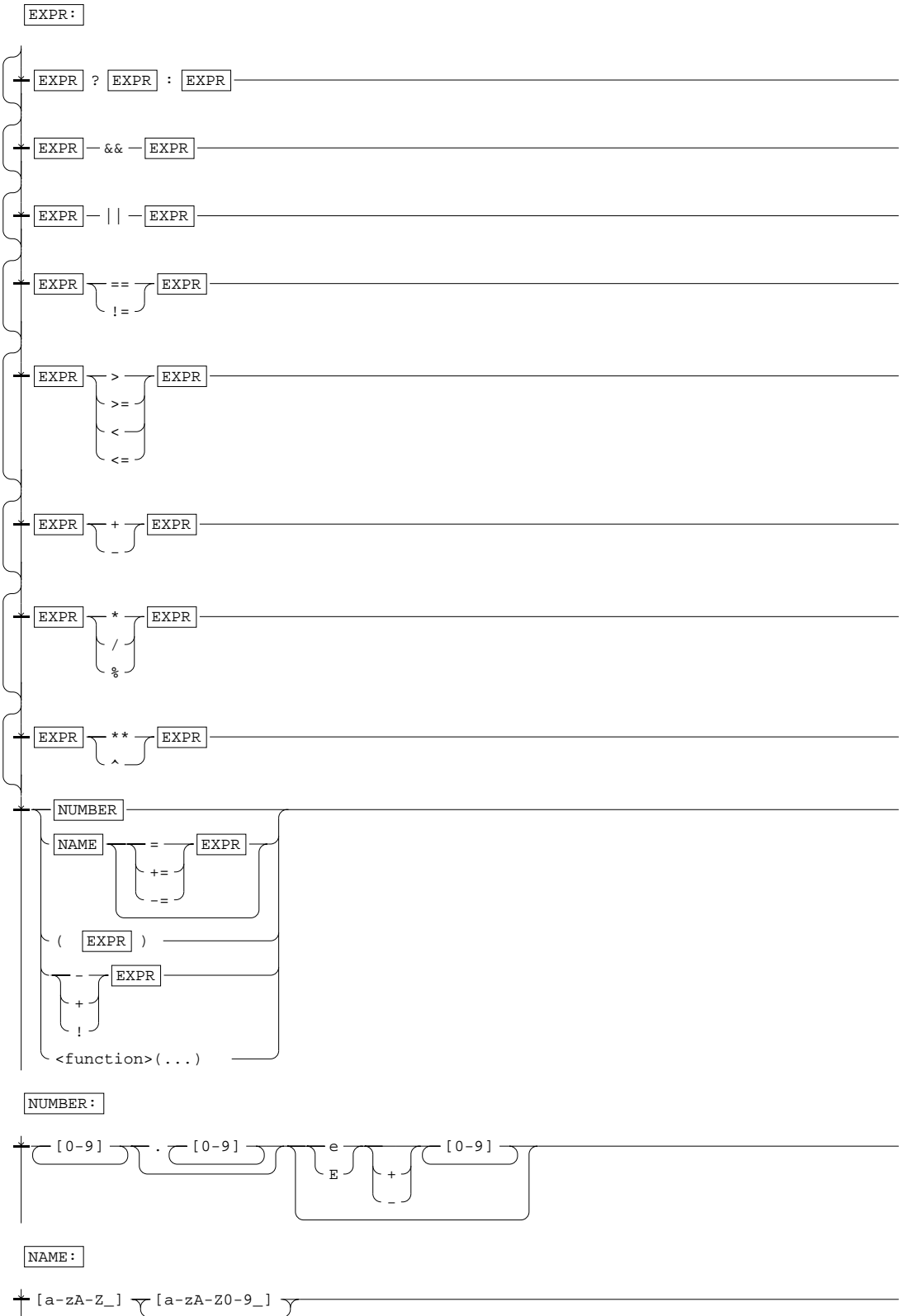
Many syntax diagrams make references to other subdiagrams in form of syntax boxes like `VAL` or `EXPR`. However, in general the syntax box is defined somewhere else in the manual and for a quick reference the following list gives you an overview of where they are defined.

<code>AT-STEP:</code> p. 80	<code>FUN:</code> p. 35	<code>PREPROCESS:</code> p. 39
<code>BC BOUND:</code> p. 46	<code>INCREMENT:</code> p. 64	<code>ROUTINE:</code> p. 36
<code>BLOCK:</code> p. 35	<code>INITIAL STAT:</code> p. 44	<code>SPEC:</code> p. 46
<code>COMMAND STAT:</code> p. 64	<code>INIT:</code> p. 64	<code>STAT:</code> p. 35
<code>COMMAND-F STAT:</code> p. 62	<code>LATTICE:</code> p. 46	<code>STRING:</code> p. 32
<code>CONVERGENCE GLOBAL:</code> p. 64	<code>LINEAR SOLVER:</code> p. 64	<code>SUBDOMAIN:</code> p. 65
<code>CONVERGENCE:</code> p. 64	<code>LITERAL:</code> p. 32	<code>UNIT:</code> p. 42
<code>DECL:</code> p. 35	<code>NAME:</code> p. 26	<code>VAL:</code> p. 25
<code>EXPR:</code> p. 26	<code>NUMBER:</code> p. 26	<code>VAL-N:</code> p. 27
<code>EXPR-N:</code> p. 27		

### 3.1 Numerical Values

All numerical values in the input files can be specified as algebraic expressions, whose syntax is described by the following diagram. We always read the longest possible expression and at the end an optional comma can be inserted to mark the end, sometimes a requirement in order to distinguish multiple input values.





The user has the possibility to define in-line variables with the statement `NAME = EXPR`. These in-line variables have a global character and once declared, they can

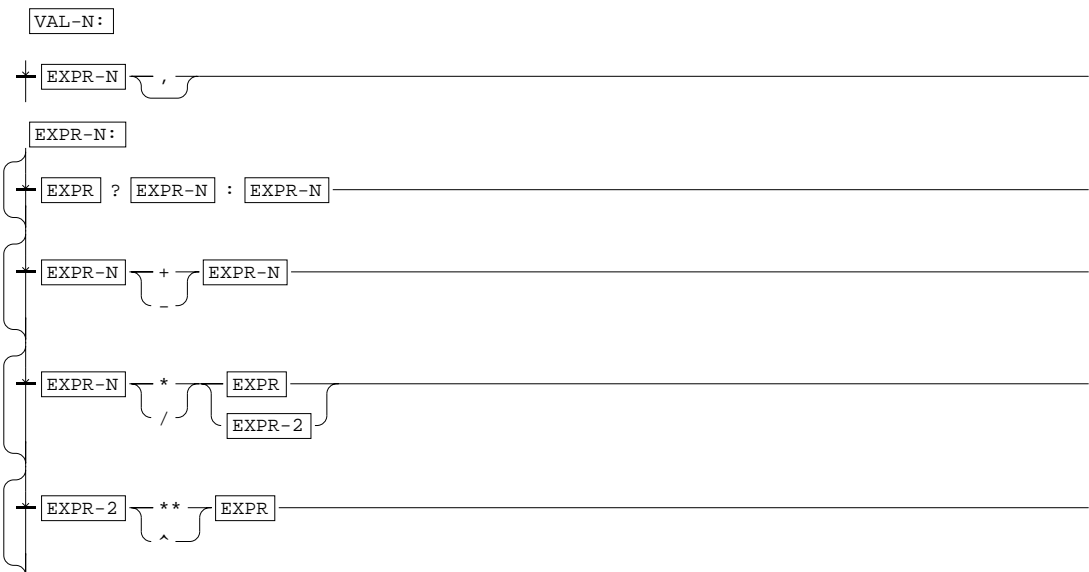
be referenced at any time. The variable value can be changed by newly defining the variable; in this case the old value is lost. However, in order to avoid inconsistencies in parallel computational environments, sometimes the global definition of variables is not available and whenever possible it has to be replaced by local variables discussed in the Section 3.5 *Block Functions* or by global definitions as presented in the Section 3.2 *User Parameters*. We discourage the in-line definition of global variables, since it may be that one is actually defining a global function which can then be called by the system at any time later on, thus also updating the global variables involved.

## WARNING !

Since we always read the longest possible expression, it is important to remember that if several numerical values must be specified sequentially, the unitary operator `-` cannot be used as the first character except for the first value. In this case, the negative values must be given in parenthesis or a comma must be used to separate the values. This is necessary in order to avoid unwanted subtraction operations reducing the number of input values.

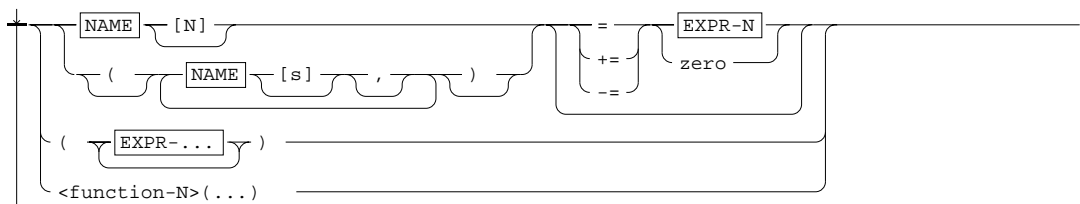
```
(* example *)
1  -1  1  (* these are two values:  0,  1      *) (* ATTENTION *)
(1) (-1) (1) (* these are three values: 1, -1, 1 *)
1,  -1,  1  (* these are three values: 1, -1, 1 *)
```

Whenever multiple input values are requested, algebraic vector expressions are also supported as described by the following diagram with  $N$  representing the number of input values and the suffix `-N` a dimension of  $N$ . The single components of the array `[NAME]` can be accessed as scalar variables using the selection operator `[s]` and  $0 \leq s < N$ . If  $s$  does not evaluate to a constant, index bounding check will be done each time the variable is accessed. The variable dimension `[N]` must only be specified by the definition e.g `a[2]=zero`, `...`, `a=1,2` and `zero` is a shortcut to zero initialization. The case with  $N=2$  can also be used to work with complex numbers since additionally it has defined the complex multiplication, division and real exponentiation, see also Table 3.2.



Symbol	Dim	Tensor components	Description
<T1> <T1Z>	3D	x, y, z	rank 1 tensor i.e. a vector.
<T1>	2D	x, y	2D version
<T1Z>	2D	x, y, z	2D version with z-component
<T2> <T2Z>	3D	xx, yy, zz, xy, yz, xz	rank 2 symmetric tensor
<T2>	2D	xx, yy, xy	2D version
<T2Z>	2D	xx, yy, xy, zz	2D version with z-component
<T2U> <T2UZ>	3D	xx, xy, xz, yx, yy, yz, zx, zy, zz	rank 2 unsymmetric tensor
<T2U>	2D	xx, xy, yx, yy	2D version
<T2UZ>	2D	xx, xy, yx, yy, zz	2D version with z-component
<T3> <T3Z>	3D	x<T2>, y<T2>, z<T2>	rank 3 tensor symmetric in the second and third index
<T3>	2D	x<T2>, y<T2>	2D version
<T3Z>	2D	x<T2Z>, y<T2Z>	2D version with z-component
<T4> <T4Z>	3D	xxxx, xxyy, xxzz, xxyy, xxyz, xxxz, yyyy, yyzz, yxyx, yyyz, yyxz, zzzz, zzxy, zzyz, zzxz, xyxy, xyyz, xyxz, yzyz, yzxz, xzxz	rank 4 tensor symmetric in the first and second pair of indices
<T4>	2D	xxxx, xxyy, xxyy, yyyy, yxyx, xyxy	2D version
<T4Z>	2D	xxxx, xxyy, xxyy, xxzz, yyyy, yxyx, yyzz, xyxy, xyyz, zzzz	2D version with z-component

Table 3.1: Component names for tensors.



To improve the readability of array variables e.g. `a[1]`, it is possible to use the dot notation followed by a component name e.g. `a.second`. In this case, instead of defining the array dimension, one defines a list of names specifying the single components e.g. `a[first second]=1,2`. By defining a single name instead of a list, the array will have the same dimension and components of the item associated with that name, e.g. a global or material model parameter discussed on p. 54. As an example, Table 3.1 lists the components used for input values which are physical tensors.

For the scalar case, the user may recognize the similarity of the syntax with the expression syntax of the C-programming language. In order to simplify the notation, the operator precedence is not given by nested diagram boxes but by the order, with

Operator	Associativity	Description
<code>EXPR ? EXPR-N : EXPR-N</code>	Right To Left	Conditional
<code>EXPR    EXPR</code>	Left to Right	Logical OR
<code>EXPR &amp;&amp; EXPR</code>	Left to Right	Logical AND
<code>EXPR != EXPR</code>	Left to Right	Inequality
<code>EXPR == EXPR</code>	Left to Right	Equality
<code>EXPR &lt; EXPR</code>	Left to Right	Less than
<code>EXPR &lt;= EXPR</code>	Left to Right	Less than or equal to
<code>EXPR &gt; EXPR</code>	Left to Right	Greater than
<code>EXPR &gt;= EXPR</code>	Left to Right	Greater than or equal to
<code>EXPR-N + EXPR-N</code>	Left to Right	Addition
<code>EXPR-N - EXPR-N</code>	Left to Right	Subtraction
<code>EXPR-N * EXPR-1,2</code>	Left to Right	Multiplication
<code>EXPR-N / EXPR-1,2</code>	Left to Right	Division
<code>EXPR % EXPR</code>	Left to Right	Modulo Division
<code>EXPR-1,2 ** EXPR</code>	Left to Right	Exponentiation
<code>EXPR-1,2 ^ EXPR</code>	Left to Right	Exponentiation
<code>! EXPR</code>	Right to Left	Logical Negation
<code>- EXPR</code>	Right to Left	Unary minus

Table 3.2: The algebraic operators for scalar `EXPR`, vector `EXPR-N` and complex `EXPR-2` values. The operators are listed in order of increasing precedence, grouped operators have the same precedence.

highest precedence at the top and lowest precedence at the bottom of the diagram. Inside each diagram station, the precedence is the same and the associativity is *Left to Right* for the binary operators and *Right to Left* for the other operators as detailed in Table 3.2.

```
(* example *)
a / b / c      (* is evaluated as ( ( a / b ) / c )      *)
a + b * c      (* is evaluated as ( a + ( b * c ) )      *)
a ? b : c ? d : e (* is evaluated as ( a ? b : ( c ? d : e ) ) *)
```

A large variety of built-in symbols in form of constants, parameters, functions both as scalar or vector valued may be defined by the application with function arguments freely configurable. In general all built-ins are read-only symbols and just in some occasions some of them may have write access enabled. Some definitions may be dynamic and enabled/disabled on the fly depending on the input values being read. The constants and functions always defined are given in Table 3.3. Other functions may be defined by the user, see the Section 3.6 *User Routines* and some special purposes built-ins are presented in Section 3.4 *Classes of built-in parameters and functions*.

Constants & Functions	Description
DIM=2 or 3	The domain's dimension.
PI=3.1415926535897932385	Mathematical constant $\pi$ .
EPS0=8.85418 $\times 10^{-12}$ C/(V m)	Vacuum dielectric constant $\epsilon_0$ .
MUE0=4 $\pi \times 10^{-7}$ V s/(A m)	Vacuum induction constant $\mu_0$ .
Q0=1.602176 $\times 10^{-19}$ C	Elementary charge $q_0$ .

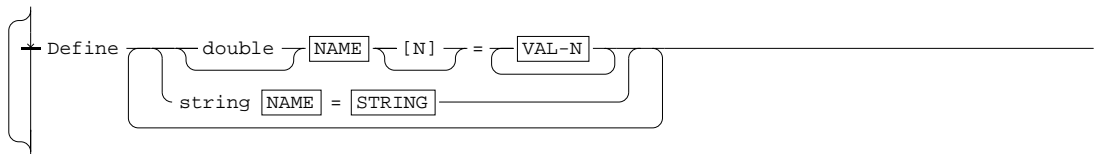
CLIGHT=2.99792458×10 <sup>8</sup> m/s	Speed of light $c$ .
HPLANK=6.62617×10 <sup>-34</sup> J s	Planck constant $h$ .
KBOLTZMAN=1.38066×10 <sup>-23</sup> J K	Boltzmann constant $k$ .
RGAS=8.314472 J/(mol K)	Universal gas constant $R$ .
AVOGADRO=6.022142×10 <sup>23</sup> 1/mol	Avogadro's constant $N_A$ .
WSIZE=32 or 64	The address bus size.
OSYSTEM	A value of 1, 2, 3 on a Windows, Linux or Mac OS.
NBLOCK, NELMT, NFACE, NEDGE	The actual number of defined macro element blocks, mesh elements, faces and edges.
NODERIV, RESTART, SOLVE, REMESH	Control variables, see p. 56, 66, 80, 79.
FAILURE	The error code after a numerical error has occurred. The value is 1 if the <code>failure</code> routine was called by the user and larger values are set for system errors.
zero	Zero value. Just as rhs of a variable assignment of any dimension.
abs, round, sqrt exp, log, loglp, erf, erfc, sin, asin, sinh, asinh, cos, acos, cosh, acosh, tan, atan, tanh, atanh( <u>VAL</u> )	Standard mathematical functions of a real value. Any functions found in standard numerical libraries can be readily made available.
cothr( <u>VAL</u> )	1/tanh( $x$ ) - 1/ $x$ : renormalized coth.
min, max, atan2( <u>VAL</u> <u>VAL</u> )	Standard mathematical functions of two real values.
exp, log, sqrt( <u>VAL-2</u> )	The overloaded version for complex numbers returning a 2-dimensional array.
norm, maxnorm( <u>NAME</u> )	Return the Euclidian and max-norm of the vector <u>NAME</u> .
finite( <u>VAL</u> )	Return one if value is neither infinite nor <i>not-a-number</i> , otherwise zero.
isnan( <u>VAL</u> )	Return one if value is <i>not-a-number</i> , otherwise zero.
random()	Return a random value between 0 and 1.
gaussian()	Return a normal Gaussian distribution.
defined( <u>ITEM</u> )	Return 1,2,3,4,0 whether the string <u>ITEM</u> is an integer number, a real number, a variable, a macro, an undefined name.
sortidx( <u>NAME</u> )	Return the permutation index to sort the vector variable <u>NAME</u> .
fromfile( <u>LITERAL</u> )	Return the value read from the file with name <u>LITERAL</u> . ATTENTION: The file cannot be an embedded file, it is opened and closed at each evaluation and the value must be plain text.
write([File <u>LITERAL</u> ]; <u>STRING</u> ), abort( <u>STRING</u> ), failure( <u>STRING</u> )	For the Front End program these routines have no effect and always return 0, otherwise <code>write</code> prints the message defined by <u>STRING</u> . Optionally the message is appended to the file <u>LITERAL</u> . The <code>abort</code> function prints the message and then stops the Kernel program. The function <code>failure</code> is similar to <code>abort</code> but the behavior is context dependent and varies from printing and aborting up to do nothing.

<code>strcmp(<b>STRING</b>, <b>STRING</b>)</code>	Return the lexicographical order of the two strings.
<code>node([Block <b>b</b>] <b>i</b>, <b>j</b>, <b>k</b>)</code>	Return the coordinates of the <b>i</b> , <b>j</b> , <b>k</b> -node of the macro element block <b>b</b> . If the <b>Block</b> option is not used, the last accessed block is considered.
<code>bdim[DIM] (<b>b</b>)</code>	Return the dimensions of the the macro element block <b>b</b> . If <b>b</b> is missing of the last accessed block.
<code>&lt;global-spec&gt;</code>	Any global parameter, see p. 54.
<code>eigenval[DIM] (<b>&lt;T2&gt;</b>), eigenpair[DIM+DIM^2] (<b>&lt;T2&gt;</b>)</code>	These functions return the eigenvalues and eigenpairs of the symmetric matrix <b>&lt;T2&gt;</b> . The eigenvectors are stored columnwise after the eigenvalues.
<code>value, step, n_step</code>	Value, last increment and number of solutions for the actual user parameter, see Section 3.2 User Parameters.

Table 3.3: Built-in constants, values and functions always available for evaluation of input values. All values are scalar if not stated otherwise. These built-ins form the class `ClassBASE`, see Table 3.4.

## 3.2 User Parameters

In order to parameterize the modeling problem, the user can define global variables of numerical type, thereafter called user parameters, with the following syntax



The variable's evaluation takes place whenever the program executes the definition and except for their dimensions, user parameters can be changed at any time with a new definition. Except some circumstances triggered by the user, there is no hidden evaluation of user parameters, so that once defined they keep their values up to the next definition, even if meanwhile the right-hand side has changed.

Some numerical input values are constants in the sense that during parsing an evaluation is performed to determine their values which is then used up to the program's end or a full restart. Here, if user parameters are involved in the definition, we have a statical dependency which is resolved once during parsing. However, many other numerical input values are actually function definitions which are then evaluated each time they are needed. If user parameters are involved here, we have a much more flexible dynamical dependency. The user can change at any times the user parameters, thus automatically changing the values of all functions directly depending upon these parameters.

There are many applications of dynamical user parameters. Typically they are changed among several solutions to model similar but different problems of interest but more involved applications are given in the Section 4.3 Non-Linear Solution Algorithms. The `time` identifier is the only default defined user parameter. It is used as any other

user parameter and just for unstationary solutions is the only one parameter permitted to change.

### 3.3 Literals and Strings

**LITERAL:**

"..."

Literals are double quoted strings of input text concatenated together to form a single string. When defining multiple literals, you need a separating comma to avoid the automatic concatenation. A double quote within the input string must be escaped by a backslash `\` and a literal backslash must be escaped with another backslash `\\`. Other escapes are available to represent special characters of the ASCII code otherwise not available and the next most important escape sequences are `\n` (newline), `\b` (backspace), `\ooo` (octal digit `ooo`), `\xhh` (hexadecimal digit `hh`), `\a` (audible alert). The literal string can be defined over multiple input lines and new line characters are taken literally and do not need to be escaped.

**STRING(CLASS):**

( LITERAL VAL-N(CLASS) )  
NAME  
<function-string>(...)

Strings are concatenation of any literals **LITERAL**, formatted numerical data **LITERAL-VAL-N**, string variables **NAME** and functions `<function-string>(...)` returning a string as argument. Numerical data can be freely formatted into text by first defining a format string **LITERAL** followed by the definition of any numerical values of the input class `CLASS`, see the Section 3.4 *Classes of built-in parameters and functions*. To avoid open end arguments, it is possible to enclose format string and numerical values within parenthesis. The string can be ended by a comma, however the comma may belong to the **LITERAL** syntax box and therefore reading the longest possible string may be impeding as for example when passing routine's parameters. Therefore we additionally stop reading the actual string whenever we have `"...", <STOP>"...` and `"...", <STOP>("....`

The format string must conform to the ANSI C standard for the `printf` family of output routines where conversion specifiers starting with a percent `%` are defined to convert numerical values. At the present time, conversion specifiers can only be declared for `float` and `double` numerical values and if other conversions are declared unpredictable results are obtained. The numerical values to be converted must be supplied after the format string, however, no check is done on the consistency of defined conversion specifiers and supplied numerical values. For a detailed description of the conversion specifiers, we refer to any documentation of ANSI C, here we just report the most used ones. After the starting `%`, conversion specifiers for floating values have an optional number specifying the minimum width of the printed result followed by



Input class	Description
ClassBASE	The base class with the built-ins of Table 3.3, always available.
ClassME	Built-ins for macro element dependency, see Table 3.5.
ClassCOORD	Built-ins for coordinate dependency, see Table 3.6.
ClassMODELRO ClassMODEL	The classes grouping the dependency of the material laws and material parameters, see p. 55. For element fields defined with the statement ElmtFieldDef, the class ClassMODEL has read-write access, but the subclass ClassMODELRO just has read-only access, see p. 49.
ClassBMODELRO ClassBMODEL	These two classes are similar to the classes ClassMODEL, ClassMODELRO and they both have read-only access to the element fields, see p. 55. For boundary fields defined with the statement BoundFieldDef, the class ClassBMODEL has read-write access but the subclass ClassBMODELRO just has read-only access, see p. 49.
ClassINCR	Built-ins for increment updates, see Table 3.17.
ClassCONVER	Built-ins for convergence criteria, see Table 3.16.
ClassCONVER_GLOB	Similar to ClassCONVER but for global convergence.
ClassSOLVER	Built-ins for convergence criteria of iterative solvers, see Table 3.23.
ClassOUTPUT	Built-ins for output and post-processing, see Table 3.13.
ClassCOMMAND	The class collecting the user's command procedures, p. 67.

Table 3.4: The list of input classes. For each input class built-in symbols are available to define input values depending on internal computed quantities. If not stated otherwise, the built-in symbols are read-only and cannot be assigned.

an optional period and the number of decimals digits printed after the decimal point. The conversion is then defined by one of the characters `f`, `e`, `E`, `g`, `G`, representing the decimal notation e.g. 13.45, the scientific notation e.g. 1.345e1 or 1.345E1 and a mixed notation chosen as the most compact of the first two forms. For example, to print the actual simulation time, you may use the format statement "Time=%g\n" time.

### 3.4 Classes of built-in parameters and functions

As shortly documented in the Section 3.1 *Numerical Values*, the SESES application can define on the fly and depending on the input value being read, built-in internal parameters to be accessed as read-only values. These parameters are accessed through their names and input values are defined as general algebraic expressions following the syntax for numerical values given in the Section 3.1 *Numerical Values*. Also available are built-in functions, an extension of built-in parameters taking arguments, with the type and number of optional or required arguments freely configurable. Sometimes built-in functions are useful ready-to-use functions preventing the user complex definitions and may exploit internal data structures to run faster. Easily defined input functions without performance gain are generally not defined as built-in functions. All these internals are used to define material laws, boundary conditions or to tune

Built-ins of ClassME	Description
<mat-spec-const>	Any constant material parameter, see the Chapter 5 <i>Physical Models</i> .
material(name)	Material name locally defined.
domain(name)	Domain name locally defined.
block([Block n;] i0 i1 j0 j1 <b>k0 k1</b> )	ME subblock locally defined. If the Block option is not used, the last accessed block is considered..

Table 3.5: The built-in symbols of ClassME allowing to describe a macro element dependency.

Built-ins of ClassCOORD	Description
<ClassME>	All built-ins of class ClassME.
<b>x, y, z</b>	The $x, y, z$ domain coordinates in unit of m.
coord[3]	The same as $x, y, z$ but with values clustered in a vector with $\text{coord}[0]=x$ , $\text{coord}[1]=y$ and $\text{coord}[2]=z$ .
<mat-spec-coord>	Any material parameter of ClassCOORD, see the Chapter 5 <i>Physical Models</i> .
FieldAt([<dof-field>   <elmt-field>]*; at(COORD))	The value of the dof-field <dof-field> or element field <elmt-field> at the point at (COORD). The notation [ ]* stands for repeated items.
<Element Field>	The element fields defined with the statement ElmtFieldDef with read-only access.

Table 3.6: The built-in symbols of ClassCOORD allowing to describe a dependency from the  $x$ -,  $y$ -,  $z$ -coordinates.

numerical algorithms and each value belong to a single input class. Input classes are coarse subdivisions between built-in input symbols having nothing in common. Inside a single class other subdivisions are possible and not all built-ins of a class are always available to define an input value of that class. The most important input classes will be first presented in the next sections and Table 3.4 gives a list of them. Here we present the first two classes most of the time included in the other ones as a subset.

The most simple input class is ClassME displayed in Table 3.5. This class defines a dependency based on a macro element selection and it is generally used to select a domain region. At the interface between finite elements, functions in this class may be discontinuous, but since evaluation generally takes place in the interior of elements, the value is always well defined. However, there are some circumstances where the evaluation is over the element boundary and here non-unique results should be avoided. The function material(name) returns one or zero whether the material name is defined or not at the point where the function is evaluated. The material name can be a predefined material or must have been defined with the initial-statement MaterialSpec as explained on p. 54. The function domain(name) works similarly but here name is the name of a domain defined with the initial-statement Domain as explained on p. 52. The function block(Block n;i0,i1,j0,j1,**k0,k1**) returns one or zero if the point lies inside or outside the rectangular subblock of macro elements i0,i1,j0,j1,**k0,k1**. The macro element block is selected with the option Block and block number n, otherwise the last selected block is used.

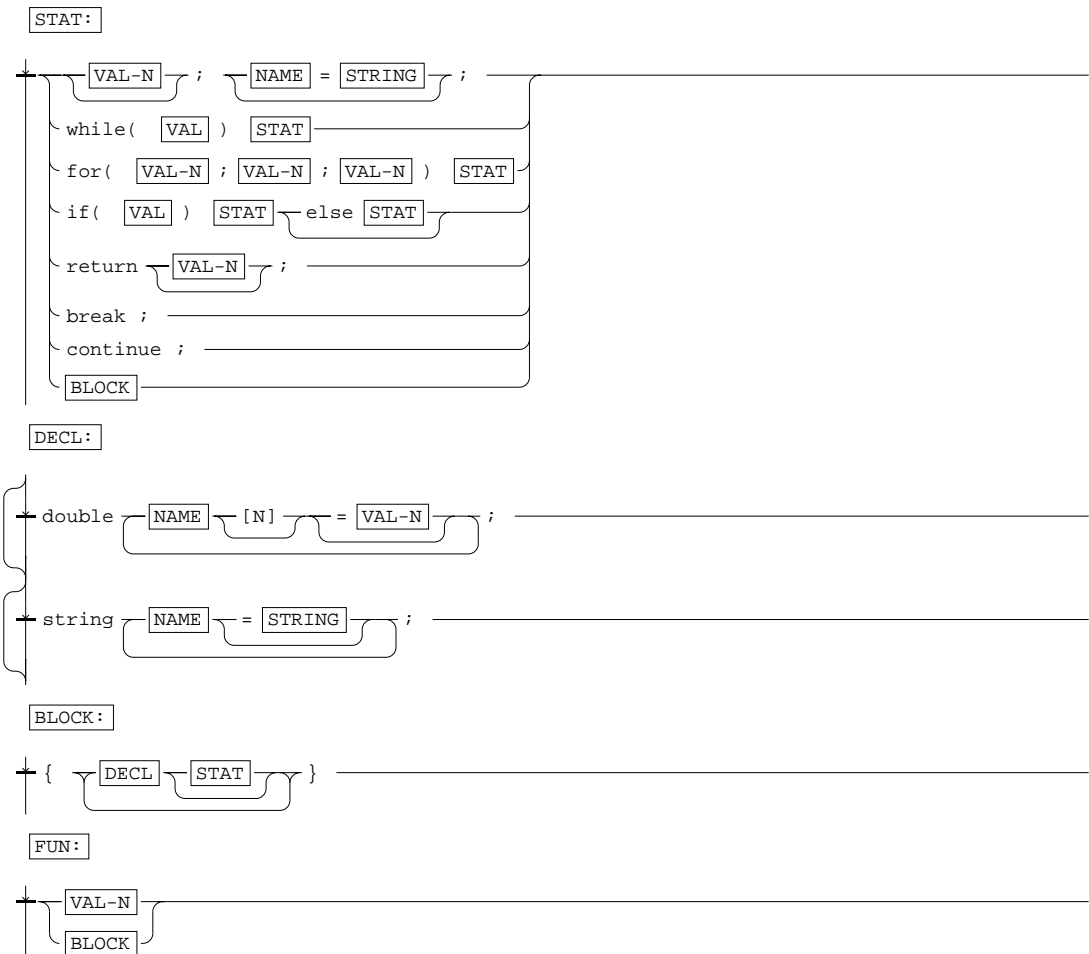
```
(* example *)
material(MyMaterial)*2 + domain(MyDomain)
```

In this example, we suppose a material `MyMaterial` and a domain `MyDomain` have been defined following the syntax diagram of the input file. On a generic point of the simulation domain, the above expression will have a value of 3, 2, 1 or 0 whether the material and the domain, only the material, only the domain or none of them are defined at that point.

The next class we present is `ClassCOORD` depicted in Table 3.6 which includes as a subset `ClassME`. For this class, the symbols `x`, `y` and `z` can be used to describe an additional dependency from the domain coordinates  $x, y, z$ . It is important to note that the built-in symbols carry a unit which must be considered when constructing expressions which in general yield values carrying a unit. At the present time this internal unit is constant and cannot be scaled. For the `x, y, z` built-in symbols of `ClassCOORD` the unit is `m`.

### 3.5 Block Functions

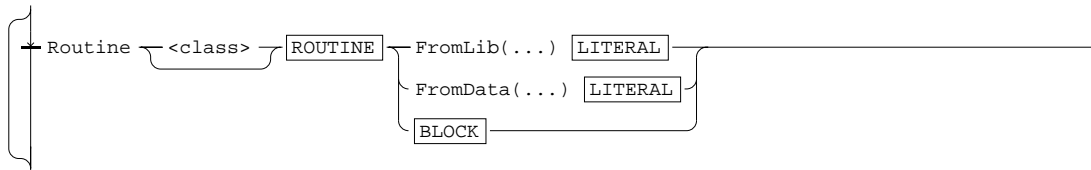
Block functions are an extension of numerical values with the possibility to define local variables and control statements.



A block is limited by a pair of braces and within the braces local variables are defined together with a sequence of expressions, string assignments and control statements following the C-programming style. The `return` statement can be used to define the routine's returned values, but most of the time and depending on the context, the returned values can be directly accessed with a name like any other variable. Returned values which are never accessed will be set to zero. If the input value is of a class listed in Table 3.4, the associated built-in parameters of this class can be accessed as any other variable.

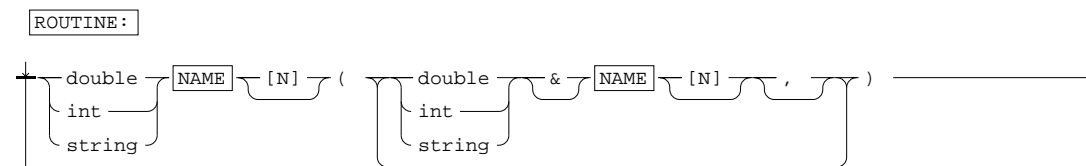
### 3.6 User Routines

Numerical routines can be defined by the user with the initial-statement



and be called when defining algebraic expressions following the syntax given in the Section 3.1 *Numerical Values*. The routine can return one or multiple values and be called with several parameters passed by value or reference. In the case the routine should access built-in parameters as discussed in the Section 3.4 *Classes of built-in parameters and functions*, these parameters can always be passed as routine parameters. However, this method has the drawback to be rather slow since all parameters must be loaded in the stack before the call takes place and error prone if several parameters are used at the same time. To improve this situation, the routine can be associated with an input class `<class>` listed in Table 3.4. In this way, all built-in parameters defined within the class can be directly accessed by the routine implementation without the need to pass them as routine parameters. However, the routine can only be called to define an input value of the same class if the input value has been enabled to depend on all accessed built-ins. The default class is `ClassMODEL` but this class is automatically changed to one of its subclasses `ClassME`, `ClassCOORD` or `ClassBASE` if built-ins are not accessed.

As first step, the routine must be declared following the next syntax diagram. Arrays of integers or strings are not available and integer values are only available with the `FromLib` option.



In a second step, the routine must be defined. If none of the `FromLib` or `FromData` options are used, the user must supply the routine implementation with a block function `BLOCK` as given in Section 3.5 *Block Functions*. By default the routine's parameters are passed by value or by reference when using `&` as prefix. Within the block

definition, the routine parameters are equivalent to local variables although with initialized values as defined by the routine call and the returned values are equivalent to local variables having the routine name.

If the `FromData` option is used, the routine can only returns double values in a vector of dimension `dim`, it must be declared with one, two or three double parameters and the name `LITERAL` of an ASCII data file must be given. These routines are always defined to be of `ClassBASE`. For one declared double parameter, each line of input data must define a parameter value and `dim` field values. The parameter values must be listed with increasing values and text can be freely appended after the field values. In the case of two declared parameters, the data file must define two integer values  $n_x, n_y$ , followed by the sorted parameter values for the first parameter  $x_1, \dots, x_{n_x}$ , for the second parameter  $y_1, \dots, y_{n_y}$  and finally the  $\text{dim} \times n_x \times n_y$  field values listed in Fortran array notation, i.e. the index associated with the first dimension is incremented first. For a routine declared with three parameters, the data file must be defined in a similar way but now for a three dimensional tensor grid. When the routine is called, the rectangular box surrounding the calling arguments is searched and the linear, bilinear or trilinear interpolation of the `dim` scalar field values at the box nodes is returned. If the calling arguments define a point outside the box of minimal and maximal values, an error message is generated. This default first order interpolation can be modified by passing options within `FromData( ... )`. The most important one is `Type DoNotInterpolate` where just the `dim` values of the box origin node are returned.

If the `FromLib` option is used, the user has to create a dynamic library named after `LITERAL` before running *SESES*. This option is to be used when the restrictions for a routine definition are too severe or one wants to optimize the routine evaluation, however, errors of any kind in the library code will inevitably result in the crash of both the Front End and Kernel programs. When creating the library, it is important to know that the routine must have the same name as declared in *SESES*, it must be of type `void routine_name(double *r)` and all the returned values followed by the routine parameters are passed in the array `r`. An integer parameter is equivalent to a double parameter properly casted. A reference parameter is equivalent to two double parameters properly casted, in the first one we store the address and in the second one the array's dimension. Here is an example of an ANSI-C source code file named `c.c` corresponding to the routine declaration: `double myroutine(double a, double &b, int c)`.

```
void /*__declspec(dllexport)*/ myroutine(double *r)
{ r[0]=r[1]+2*((double**)(r+2))[*(int*)(r+4)]; /* return a+2*b[c] */ }
```

A dynamic library named `lib.dll` can be created using the following commands, however, for Windows systems, the special declaration `__declspec(dllexport)` in the `c.c` file must be uncommented.

System	Compiler	Command
Linux	Gnu compiler	<code>cc -o lib.dll -shared -fPIC c.c</code>
Windows	VisualC++ compiler	<code>cl c.c -link -dll -out:lib.dll</code>

If global variables defined with the `Define` statement should be accessed inside the

library, they can always be passed as routine's parameters. However, to improve performances, it is better to define a routine named `callme` that will be called once to pass the pointers of the variables to the library. These variables, should be considered as read-only by the library. Here is an example of an ANSI-C source code to be added to the previous code in order to obtain the pointers `ptime` and `pnum` to the simulation parameter `time` and user defined variable `num`.

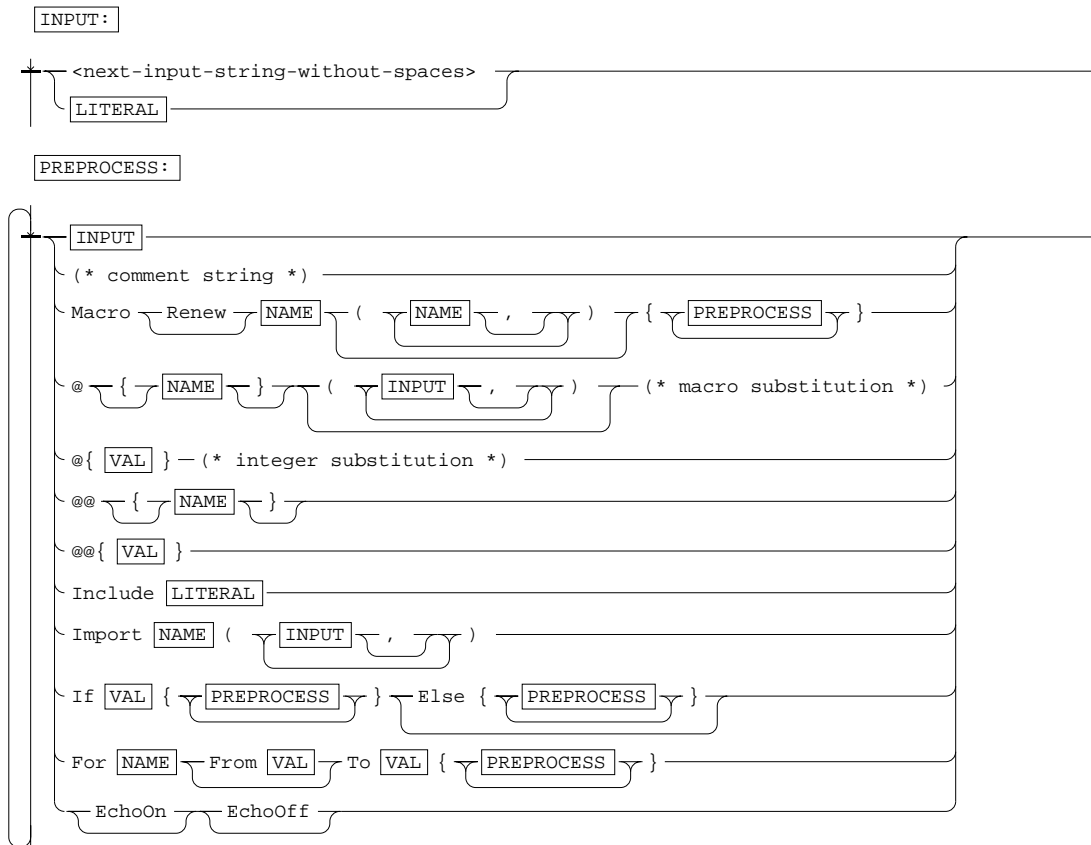
```
static const double *ptime, *pnum;
int /*__declspec(dllexport)*/ callme(const double *(*get)(const char *))
{ return !(ptime=get("time")) || !(pnum=get("num")); }
```

The routine `callme` is called once when opening the library together with the routine pointer `get` as parameter. This routine should in turn be called with the variable name to retrieve the pointer to the global variable. If the variable is not defined, a null pointer is returned. In the case all pointers have been successfully obtained, the routine `callme` should return a null value, otherwise a non-zero one.

If the routine belongs to an input class different from `ClassBASE`, the built-in symbols of the input class can also be directly accessed without the need to pass them as routine's parameters. However, some technicalities are involved here as for example the usage of some include files and the built-in dependency declaration, technicalities explained elsewhere.

### 3.7 Text Preprocessor

Similar to the C-programming language, the *SESES* input language provides certain language extensions by means of a text preprocessor. The input text is first filtered by the preprocessor before being passed to the input parser. In the following, the syntax diagram of the preprocessor is presented.



The **VAL** box represents a numerical value. Here the user has the possibility to define the value using an algebraic expression, as described in the Section [3.1 Numerical Values](#). Although not explicitly given in the syntax diagram, the input text used to evaluate the **VAL** box is itself preprocessed but only for comments (`* . . . *`) and substitutions `@ . . .`.

### Preprocessor-statement (`* . . . *`)

Comments can be placed anywhere in the input text and they have the highest precedence i.e. they are removed from the input text before any other preprocessing task takes place. Together with the main issue of providing additional information inside the input text, comments are very useful to switch on and off portions of the input text. Therefore, comments can be nested to any level but they cannot cross file boundaries i.e. comments must begin and end inside the same file. Comments are equivalent to a space separator, so that the single string `a ( ** ) b` is converted to `a b`.

### Preprocessor-statement Macro

This preprocessor statement defines the macro **NAME** with possibly any number of parameters declared within the parenthesis and separated by a comma. The body of the macro is the text defined within the braces and the body will be substituted each time the macro is referenced in the form `@name` or `@{name}`. If the macro defines



parameters, they must be defined when referencing the macro and their value within the body is accessed in the form `@par` or `@{par}` where `par` is the defined macro parameter. If multiple parameters are defined, they must be separated by a comma whereas commas within literals or paired parenthesis do not count. The start and the end of a macro always correspond to a space separator, however this is not the case for macro parameters so that here textual concatenation is possible in the form `A@par1@{par2}D` resulting in e.g. `ABCD` if the macro parameters `par1`, `par2` have values `B` and `C`. Macros can be newly defined with the option `Renew`.

### Preprocessor-statement `@ . . .`

This preprocessor statement performs the substitution of the argument following `@` which may be a macro, a macro parameter or an expression to be evaluated and converted to a textual integer value. The argument may be optionally enclosed in a pair of braces. This is required if the expression is not a single variable or if one wants to preprocess the text within the braces for comments and substitutions. For macro parameters substitution and integer substitution, the operator `@` can be used to form strings by concatenation in the form e.g. `a_@{1+1}_b=exp(@{1+0.3})` resulting in the string `a_2_b=exp(1)`.

The integer substitution is mostly used together with the `For-loop` preprocessor statement and here we warn the user about some side effects. Although some variables have been defined, they may be unavailable because they have not been evaluated. As an example, the expression `a=1, @a` leads to an error. This is because the input text is passed through the preprocessor as a single string but the preprocessor itself does not evaluate the variable `a` and thus it cannot evaluate the expression `@a`. Differently the expression `a=1, @a` is correct. When the string `@a` is preprocessed, the expression `a=1` has already been read and evaluated by the input parser and therefore the variable `a` is available.

### Preprocessor-statement `@@ . . .`

This preprocessor statement is similar to the previous one but produces double quoted strings and the parameter following `@@` can only be a macro parameter or an expression to be evaluated to integer. A macro parameter is taken literally and it is not recursively evaluated for substitutions.

### Preprocessor-statement `Include`

This preprocessor statement can be used to maintain a library of input file fragments for multiple uses. The preprocessor will replace the include statement with the contents of the file specified by `LITERAL`. If a container file has been specified, see the Section 1.6 *Embedding files in a single container file*, the preprocessor looks as first option inside the container file for the file to be included. If this option fails or does not apply, the file must be resident in a directory specified by the `search-path` command inside the input file. The program working directory is always included in the `search-path` as



well as the directories `BIN`, `BIN/../../include` and `BIN/../../../include` with `BIN` the directory of the installed *SESES* binaries.

### Preprocessor-statement **Import**

For complex input tasks, one may think to use external programs to create input statements which written to some files are included with the `Include` statement. This statement works similarly, but it is more dynamic and flexible in the sense that it calls a library routine with any number of user parameters to obtain the input text. Prior to use this statement, a routine `NAME` must be declared following the `ROUTINE` syntax box, see p. 36, with a returned value of type `string` and the routine being loaded from a library with the option `FromLib`. The library routine must store the input text in the memory returned by the `string` and the text must be null-ended. If the text's length in the returned string is set to some non-zero value, the memory must be allocated with `malloc` and will be deallocated by *SESES*, otherwise the user is responsible for deallocating the memory.

### Preprocessor-statement **If-Else**

The `If-Else` preprocessor statement is used to switch on and off portions of text inside the input file depending on the if-else conditional value. Typically the user defines a global variable with the `Define` statement and then uses the variable to construct conditional input. This writing style has the advantage that different input files can be generated by simply changing the values of some variables.

```
(* example *)
Define with_alu = 1 (* use 0 for the default material *)
...
If ( with_alu ) { MaterialSpec Alu Parameter ... }
...
If ( with_alu ) { Material Alu 1 }
...
```

In this example, the user defines at the beginning the variable `with_alu`. If the variable is non zero, a new material `Alu` is defined and used as default material. Therefore, by merely changing the variable's value, it is possible to run two different modeling problems.

### Preprocessor-statement **For**

The `For`-loop preprocessor statement is used to repeatedly create similar portions of text. A loop variable must be defined and its scope or range of definition is limited to the `For`-loop only. The text within the braces is repeated for values of the loop variable starting from the `From` value and up to the `To` value by using increments of one. If not defined, the default start value is zero. The loop variable can be used to create different text at each loop iteration, however, it must be referenced with the operator `@` or `@{ }` previously described as they were macro parameters.



Unit	Symbol	Keyword
meter	m	m
second	s	s
gram	g	g
Coulomb	C	C, Cb
Kelvin	K	K
Newton	N	N
Joule	J	J
electron Volt	eV	eV
Volt	V	V
Ampere	A	A
Pascal	Pa	Pa
Watt	W	W
Siemens	S	S
Tesla	T	T, Tesla
Henry	H	H

Table 3.7: The SESES basic units.

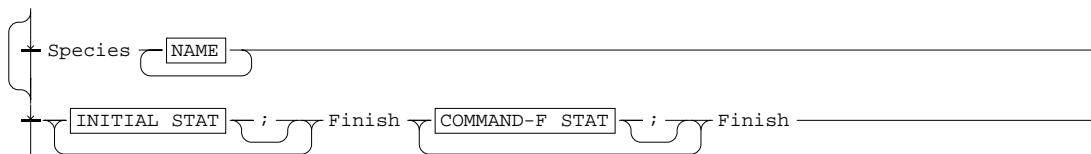
Prefix	Multiple	Keyword
peta	$10^{15}$	P
tera	$10^{12}$	T
giga	$10^9$	G
mega	$10^6$	M
kilo	$10^3$	k
deci	$10^{-1}$	d
centi	$10^{-2}$	c
milli	$10^{-3}$	m
micro	$10^{-6}$	u
nano	$10^{-9}$	n
pico	$10^{-12}$	p
femto	$10^{-15}$	f

Table 3.8: The SESES basic unit prefixes.

only once, if followed by the repetition operator #n with n the number of repetitions. Similarly, if a group of units is repeated  $n$ -times, one can use the operator ##n to define the group  $n$ -times which applies the repetition from the beginning or if defined, from the last occurrence of the operator ##.

SESES does favor the SI units m, s, kg, C, K and their compounds without any other prefixes in the sense that if these SI units are used no internal conversion factors are defined. As a shortcut to use these SI units, the user may use SIunit to bypass the unit definition and thus to define a conversion factor of one.

### 3.9 Input Syntax



The SESES input file is divided into an initial section and a command section. Both sections are made of a series of statements of type `INITIAL STAT` and `COMMAND-F STAT` in any arbitrary order and terminated by a `Finish` symbol. The initial section serves to describe the modeling problem like the macro element mesh, the equations to be solved, the boundary conditions and the physical models. The command section specifies the numerical algorithms like the solver type used to solve the linearized and discretized governing equations, the non-linear solution algorithm and the generation of graphical data to be visualized by the Front End program. The initial section is based on a static approach with data easily visualized within the Front End program, whereas the command section uses a more flexible dynamical and procedural

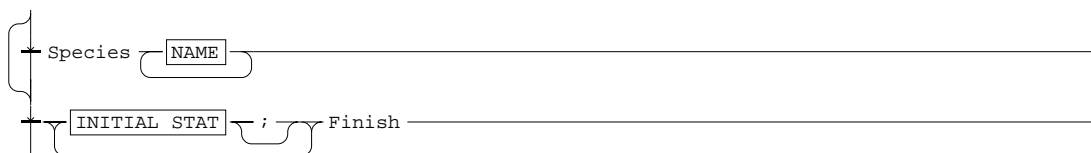
Macro	Description
FILEDIR	The directory path of the input file as literal.
BINDIR	The directory path of the SESES binaries as literal.
VERSION	The version tag of the SESES binaries as literal.

Table 3.9: Built-in macros always available to define textual input.

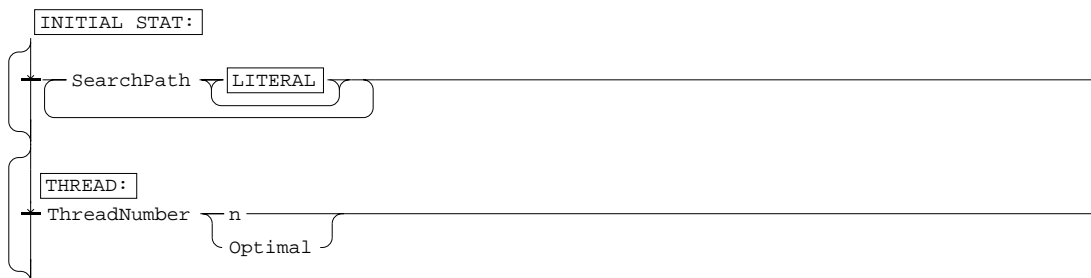
approach.

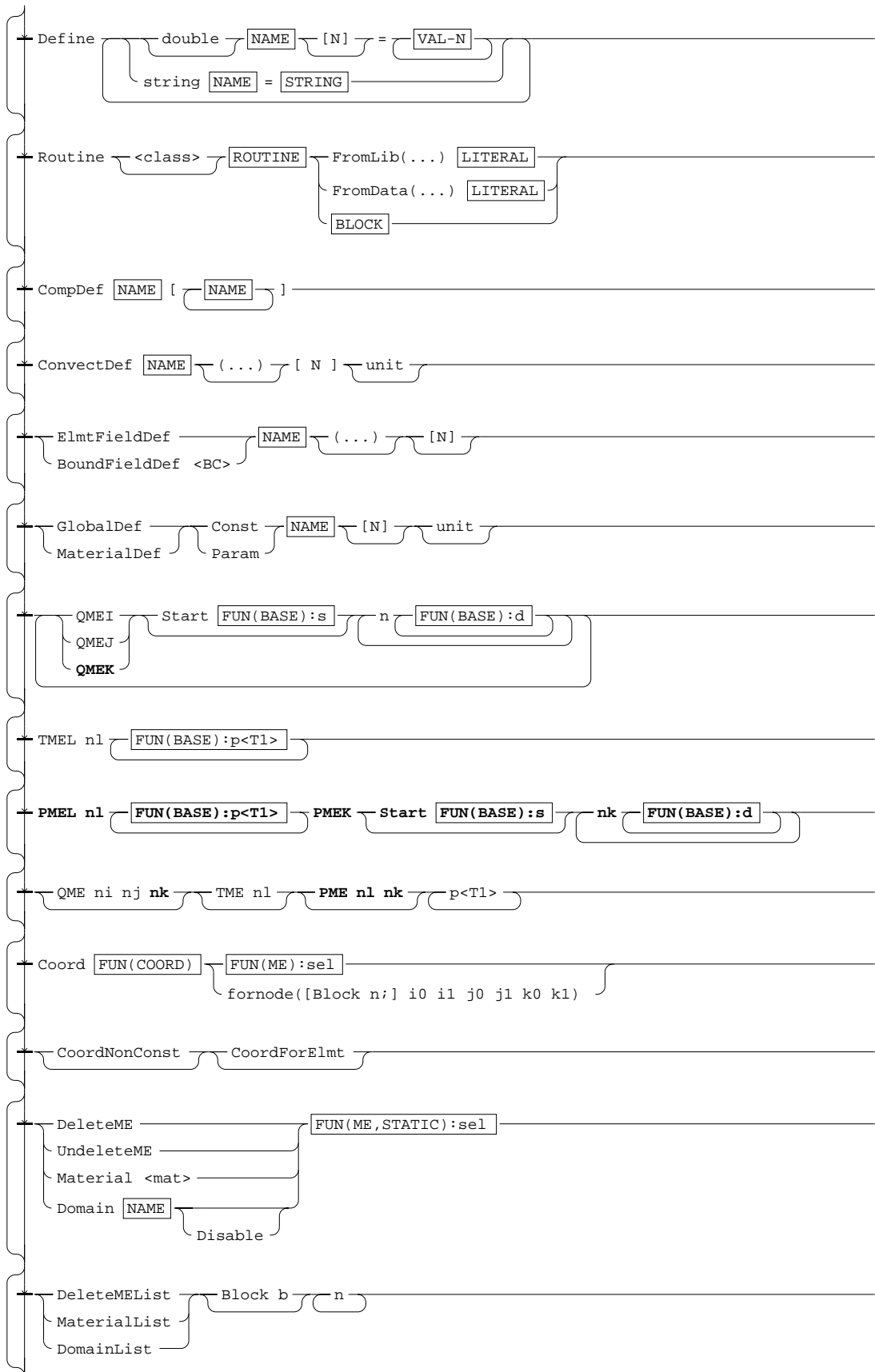
To specify a numerical value, all built-in constants and functions of the input class `ClassBASE` as given in Table 3.3 are available. All numerical values not displaying an input class dependency are constant values of type `VAL` just evaluated when parsing the input. This is also true for functions flagged as `STATIC` in the class dependency. Therefore any dependency of these constants or static functions from user parameters, as explained in Section 3.2 *User Parameters*, is not considered anymore after parsing. Items in **bold** are allowed in the 3D-version only. Non-numerical built-in values are defined by built-in macros, listed in Table 3.9.

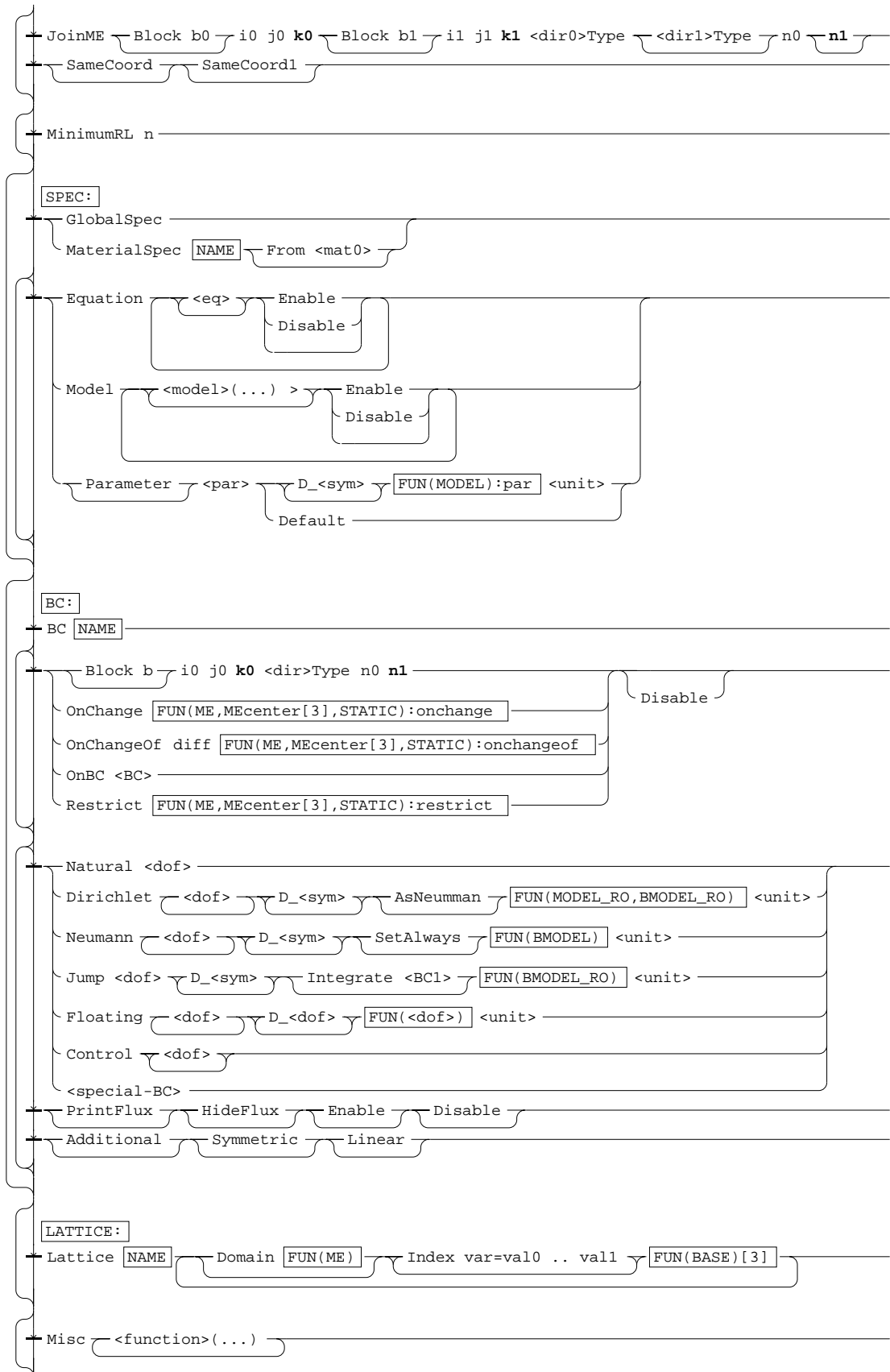
### 3.10 Syntax of the initial section



The initial section starts with the `Species` statement to set up the parser and afterwards follows a series of statements of type `INITIAL STAT` in any arbitrary order and terminated by the `Finish` symbol. However, the statement order is important since generally it gives different results and also some statements may require data defined beforehand by other ones. The statements can be optionally ended by a semicolon, sometimes a necessity in order to stop reading the actual statement if the following part may belong both to the actual and next statement. The initial section describes an initial and static configuration easily visualized by the Front End program and here text-preprocessing as described by the Section 3.7 *Text Preprocessor* is mainly used to construct complex structured input files.







## Initial-statement Species



When modeling mass transport with several species, the definition of the species together with their kinetics is completely free and is part of the problem specification. However, for user friendliness many input symbols have been defined to depend from the species names and these names must be known at the very beginning when parsing the input files which is done with this preamble statement. For the available physical models related to the defined species, see the Chapter [5 Physical Models](#).

## Initial-statement SearchPath



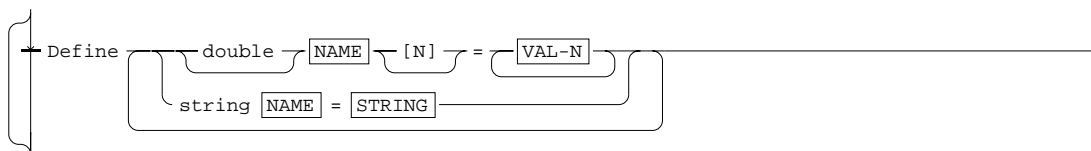
The `SearchPath` statement enables the preprocessor to search for files in the directory `LITERAL` additionally to the current directory of the running process as well as the directories `@BINDIR`, `@BINDIR"/../include"`, `@BINDIR"/../../include"` with `@BINDIR` the directory of the installed *SESES* binaries, see Table [3.9](#). If multiple directories are specified, in order to avoid the automatic concatenation of literals, you need either several statements or a separating comma. The latest specified directories are searched first.

## Initial-statement ThreadNumber

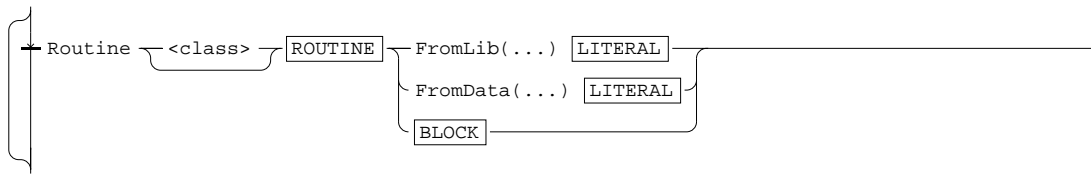


For concurrent processing, *SESES* supports the shared memory model with forking threads and this statement defines the number of threads `n` used to run parallelized code. The Kernel program option `-t <n>` can also be used to set an initial value for `n`. For optimal speed-up, the number of threads should be the same as the number of processors which can be set with the option `Optimal`.

## Initial-statement Define Routine







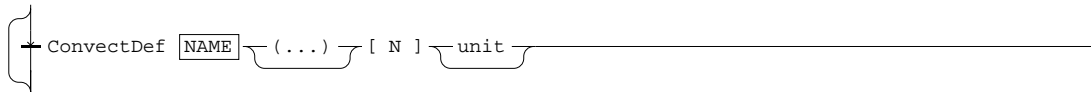
These statements define global variables i.e. user parameters and routines, and since they represent general concepts they have already been described in the Sections [3.2 User Parameters](#) and [3.6 User Routines](#). Because of the static nature of the initial section, the definition and evaluation of the user parameters by the `Define` statement is done just once and in a strictly sequential manner up to the end of the initial section.

### Initial-statement `CompDef`



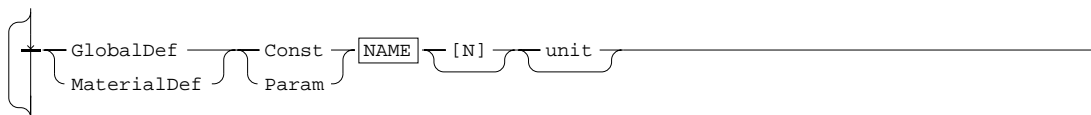
The statement `CompDef` collects together the definition of component's names for vectors additionally to the ones statically defined by Table [3.1](#). The first name is the new vector type which is followed, enclosed in brackets, by  $n$ -names for each component. Vectors of  $n$ -dimension with these named components can then easily defined with the vector type name, see also the Section [3.1 Numerical Values](#).

### Initial-statement `ConvectDef`



The statement `ConvectDef` declares the dof-field `NAME` of dimension  $N$  and unit `UNIT` to be a convected field along the streamline of a vector field. Within parenthesis, one can additionally specify the name and unit of the associated flux. The equation `Equation Convect<name>` is then available to solve for the dof-field `<name>`.

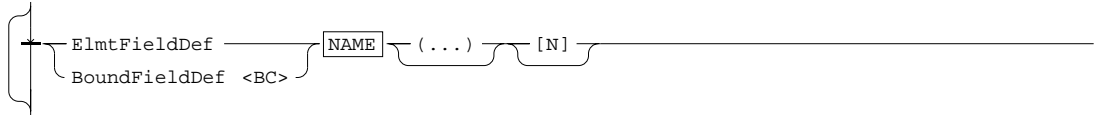
### Initial-statement `GlobalDef` `MaterialDef`



These statements allow the user to add parameters to the global and material data bin presented on p. [54](#). For global parameters, the option `Const` is mandatory. However, for material parameters there is the choice among a constant parameter `Const` to be accessed within the input class `ClassME`, see Table [3.5](#) and a model parameter `Param` to be accessed within `ClassMODEL`, see Table [3.10](#). The parameter's name `NAME` then follows with optionally the dimension and unit specification. Even when using the `Const` option, one is always defining functions which are evaluated as needed and

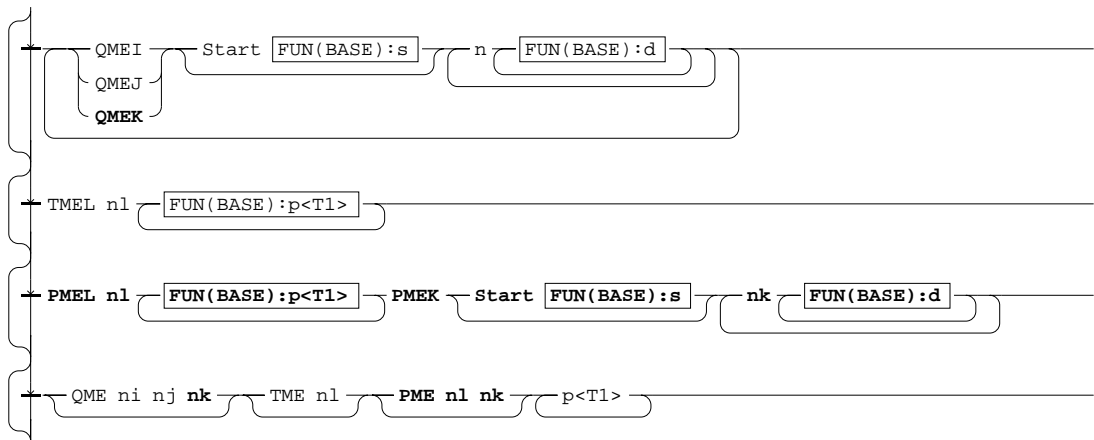
this is the major difference with user parameters just evaluated when defined, see also the Section [3.2 User Parameters](#).

### Initial-statement ElmtFieldDef BoundFieldDef



These statements declare a global element or boundary user field named after `NAME`, i.e. a scalar or vector field of dimension `N`, defined on the domain or on the boundary `<BC>` with field values stored in resident memory and accessible in read-write mode by the user. User fields are typically used to implement material laws which are history dependent and therefore require global storage of information among different solution steps. Everything declared here but not defined later on, will have a default zero value since no memory is allocated at this time. The memory type of a user field can be given here by specifying the options of Table [3.26](#) within parenthesis or later on with the `Store` statement explained on p. [86](#) when the user field is actually defined and allocated. For element fields, there is read-write access with the class `ClassMODEL` and read-only access with the classes `ClassMODEL_RO`, `ClassBMODEL`, `ClassBMODEL_RO` and for boundary fields, there is read-write access with the class `ClassBMODEL` and read-only access with the classes `ClassBMODEL_RO`, see Tables [3.10-3.11](#). However, if read-write access is enabled, the sets of points for definition and evaluation must agree.

### Initial-statement QME TME PME <shape>ME<dir>



These statements define the initial mesh where all geometrical lengths involved must be given in unit of `m`. Differently from model parameters where units are always required, for geometrical lengths which are often defined, we avoid forcing the declaration of the unit `m`.

The computational mesh is formed by blocks of macro elements having the form of a (3D) hexahedron, (3D) tetrahedron or (3D) prism and (2D) rectangle or (2D) triangle,

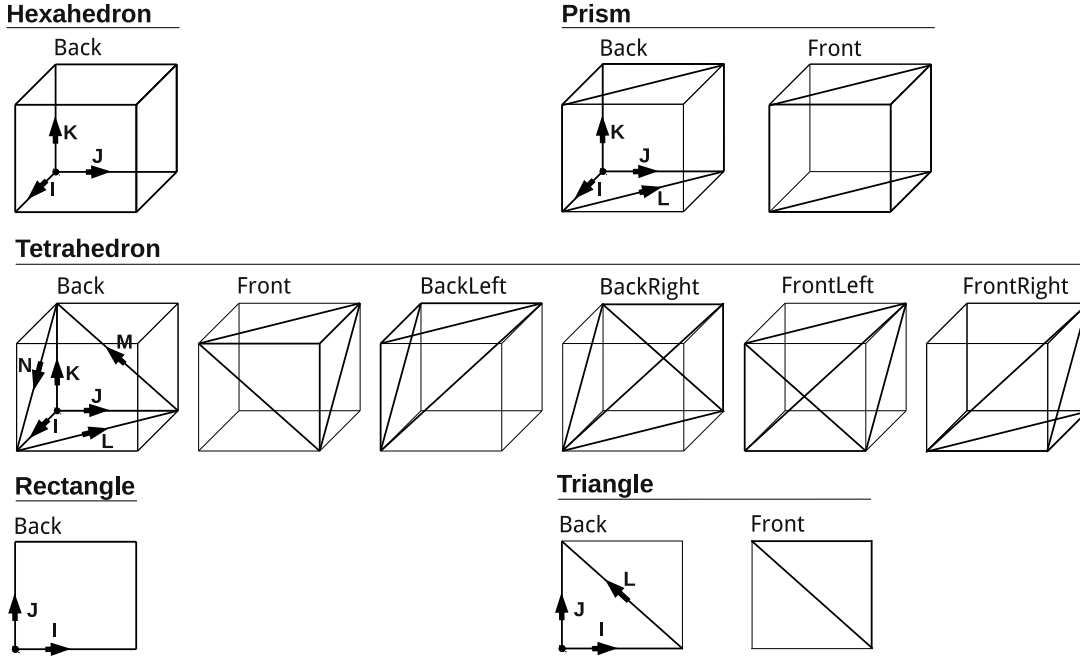


Figure 3.1: Subdivision of macro element block cells and block directions.

as shown in Fig. 3.1. Along the block  $I$ -,  $J$ - and  $K$ -directions, we perform a tensor-product subdivision in cubic cells. Each cell is then decomposed into one hexahedral, six tetrahedral, two prismatic, one rectangular and two triangular macro elements whereas macro elements not belonging to the block shape are removed. With the initial-statements `DeleteME` and `JoinME` described later, macro elements of a block can be deleted and macro elements belonging to different blocks can be arbitrarily connected together. Macro elements are used to define an initial mesh fine enough to specify material regions, boundary conditions or any other region of interest. For computational purposes this initial mesh may be too coarse, however, hexahedral and rectangular macro elements can be locally and adaptively refined into smaller finite elements, whereas tetrahedral, prismatic and triangular macro elements support at the present time just a homogeneous refinement. This process allows to create computational meshes adapted to the required numerical accuracy. Mesh refinement is controlled with the `Remesh` statement explained on p. 79.

For the basic shapes specified by the prefixes  $Q$  for the (3D) hexahedron and the (2D) rectangle,  $P$  for the (3D) prism and  $T$  for the (3D) tetrahedron and (2D) triangle, two statements are available for defining a block of macro elements. The first one is a short form with limited possibilities for defining single macro element shapes, whereas the second one is generic. Per default, the first defined macro element block receives the block number zero; all other blocks will be numbered with increasing numbers by following the order of definition. It is important to remember the definition order for blocks i.e. the block number, since many initial-statements require the block number as input value. The global variable `NBLOCK`, see Table 3.3, returns the number of actual defined blocks and may be used to label blocks, since its value represents the block number of the next defined block.

With the first short form of the macro element block definition `<Shape>ME<Dir>` and

for the (3D) hexahedron, the (2D) rectangle or the third dimension of the (3D) prism, the user specifies the number of block macro elements and their size along the block I-, J- and K-directions which initially correspond to the  $x$ -,  $y$ - and  $z$ -axes. If  $n$  is positive, we have  $n$  macro elements with the same side length of  $d$ , whereas if  $n$  is negative, we have  $|n|$  macro elements with different side lengths specified by the following  $|n|$ -values of  $d$ . With the `Start` option, it is possible to define the starting value  $s(\text{BASE})$  for the first block coordinate. For the (3D) tetrahedron, the (2D) triangle and the first two dimension of the (3D) prism, the user specifies the number of block macro elements along the block L-direction and the world coordinates  $p<\text{T1}>$  of the three or four nodal points. An equidistant subdivision is then applied for the other block directions.

The shape of any macro element can later be changed with the `Coord` statement. To save computational resources, by default the mesh geometry is taken as constant and the functional dependency of  $s(\text{BASE})$  and  $d(\text{BASE})$  from user parameters is not considered anymore after parsing. To allow a dynamic dependency from user parameters of the initial mesh geometry, the optional statement `CoordNonConst` need to be used.

With the second form of the macro element block definition `<Shape>ME`, the user supplies the number of block macro elements along the block I-, J- K- and L-directions and the world coordinates  $p<\text{T1}>$  for each node of the macro element block. For a (3D) hexahedral block with  $n_i, n_j, n_k$  macro elements in the I-, J- and K-directions,  $(n_i + 1) \times (n_j + 1) \times (n_k + 1) \times 3$  node coordinates must be given following the order

```
for i=0 to (ni)   for j=0 to (nj)   for k=0 to (nk)       (x,y,z)
```

For a (3D) tetrahedral block with  $n_l$  macro elements in the L-direction,  $(n_l + 1) \times (n_l + 2) \times (n_l + 3)/6 \times 3$  node coordinates must be given following the order

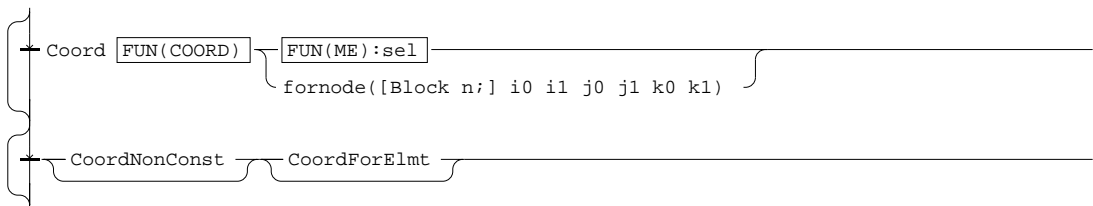
```
for i=0 to (nl)   for j=0 to (nl-i)   for k=0 to (nl-i-j)   (x,y,z)
```

For a (3D) prismatic block with  $n_l, n_k$  macro elements in the L- and K-direction,  $(n_l + 1) \times (n_l + 2) \times (n_k + 1)/2 \times 3$  node coordinates must be given following the order

```
for i=0 to (nl)   for j=0 to (nl-i)   for k=0 to (n_k)       (x,y,z)
```

For the (2D) rectangular and (2D) triangular block, proceed as for the (3D) hexahedral and (3D) tetrahedral block without considering the third dimension.

### Initial-statement `Coord` `CoordNonConst` `CoordForElmt`



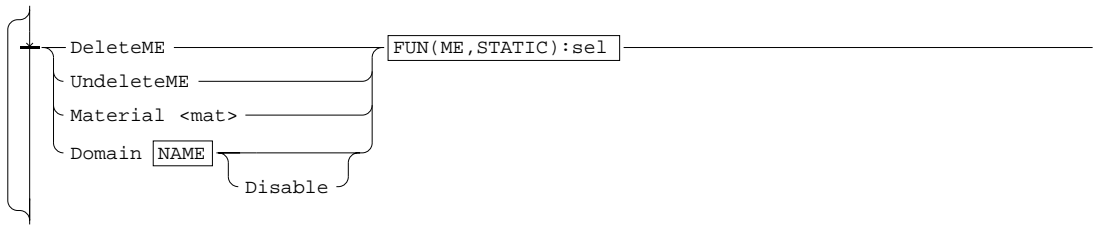
This statement changes the geometry of the macro element mesh. The new node coordinates are defined as a 2D/3D function of class `ClassCOORD` and set for all macro

elements selected by a non-zero value of the selection map `sel (ME)` defined as a function of the input class `ClassME`, see Table 3.5. For the case when just single macro element nodes, edges or faces should be defined, the special function `forNode` can be used to select the `i0-i1, j0-j1, k0-k1`-nodes of the macro element block `n`. If the `Block` option is not used, the last selected block is considered.

By default the geometry of the mesh is considered constant, so that computational resources can be saved by just performing a single initial evaluation of the geometry at the time of parsing. However, if the mesh geometry depends dynamically on user parameters, the option `CoordNonConst` must be used, see also the Section 3.2 *User Parameters*. The placement of this option is important, since just after its declaration, the back storage of the functional statements defining the geometry is started. To save memory and to gain on speed, it should be declared as late as possible and just before the first `Coord` or `QMEI` statement changing dynamically the geometry.

By default the coordinate functions are evaluated just at the macro element nodes and these values are linearly interpolated to compute the node coordinates of elements inside the macro elements. With the option `CoordForElmt`, the coordinate functions are used to compute the node coordinates of every element. This option must be specified prior any `Coord` statements, it prevents using the `forNode` function and the `CoordNonConst` option.

### Initial-statement `DeleteME UndeleteME Material Domain`



This statement maps properties onto a region of the simulation domain.

The `DeleteME` statement physically removes macro elements. In this way indentations or holes are created in the macro element mesh. The `UndeleteME` statement undoes the action of the `DeleteME` statement.

The statement `Material` maps or associates a material onto a region of the simulation domain, the material `<mat>` must have been defined with the `MaterialSpec` statement explained on p. 54.

The statement `Domain` defines subdomains of the simulation domain. The name `NAME` can be the name of a new domain or of a previously defined domain. If the `Disable` option is used, a domain is cleared instead of being defined. Domains are not used by *SESES* but just by the users in order to simplify the definition of `ClassME` functions together with the built-in function `domain`, see Table 3.5.

After selecting a property to be mapped onto the simulation domain, for each macro element the selection map `sel (ME)` of the input class `ClassME`, see Table 3.5, is evaluated and if a non-zero value is returned the property is set on the macro element. This

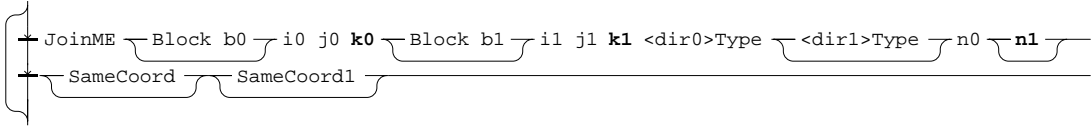
property mapping is applied just once when parsing and therefore any dependency of the selection map from user parameters, as explained in Section 3.2 *User Parameters*, is not considered anymore after parsing.

### Initial-statement DeleteMEList MaterialList DomainList



The statement `DeleteList` followed by  $n_x \times n_y \times n_z$  boolean values specifies if each macro element of the block `b` is deleted or not; if the `Block` option is omitted, the last selected block is used. Similarly, the statement `MaterialList` followed by  $n_x \times n_y \times n_z$  integer values specifies the material number associated with each macro element. The statement `DomainList` followed by  $n_x \times n_y \times n_z$  integer values in hexadecimal format specifies if each macro element belongs to a user defined domain. Each domain is represented by a single bit in the hexadecimal number with the bit order being the same as the order of the domain's definition. For all three statements, the macro element order of the integer values is the same as for specifying the coordinates with the statement `QME`.

### Initial-statement JoinME



This statement joins together macro elements along (3D) faces or (2D) edges and allows to work with generic unstructured macro element meshes. Another application is the specification of periodic boundary conditions by joining together the beginning and the end of the domain boundary.

In order to join together (3D) quadrilateral or triangular faces, as first one defines two nodes of the macro element mesh specified with the block node indices `i0 j0 k0` and `i1 j1 k1` and block numbers `b0` and `b1`; if the `Block` option is omitted, the last selected block is used. Afterwards one defines a first pair of block directions `<dir0>Type` e.g. `IJType` for the first face possibly followed by a second pair `<dir1>Type`, if different from the first one. The available block directions are given in Fig. 3.1 and depend on the block macro element shape. For a quadrilateral face, the last two integer values `n0 n1` which can be positive or negative but not zero, specify the direction to go when joining the faces along the pair of block directions. For the first face, we always consider absolute values  $|n_0|$ ,  $|n_1|$ , whereas for the second face the specified values  $n_0$ ,  $n_1$ . For a triangular face just a single integer value must be specified and we set  $n_1 = n_0$ . In order to join together (2D) edges, one proceeds similarly, but here just one block direction is defined. If the `SameCoord` option is used, the geometric coordinates of the first face are copied into the second face and reversely

for the SameCoord1 option. This ensures the consistency of the geometric data since joining faces or edges is something algebraic; they are not required to be geometrically the same boundary.

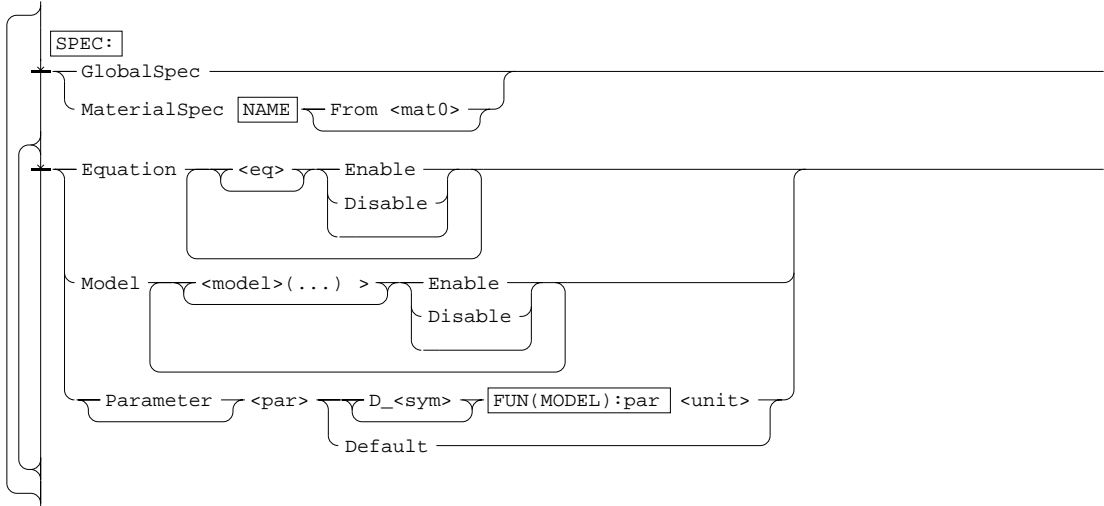
For example, for a (3D) hexahedral block and block directions JKType for both faces, the macro element nodes  $[i_0, j_0 + j, k_0 + k]$  and  $[i_1, j_1 + j \text{ sign}(n_0), k_1 + k \text{ sign}(n_1)]$  for  $j = 0 \dots |n_0|$  and  $k = 0 \dots |n_1|$  will be identical.

### Initial-statement MinimumRL



This statement can be used only once and defines the initial refinement level for the whole macro element mesh. For a specified refinement level  $n$ , each non-deleted macro element is subdivided into  $4^n$  finite elements for 2D meshes and into  $8^n$  finite elements for 3D meshes. The default refinement level is  $n = 0$ .

### Initial-statement GlobalSpec MaterialSpec



These two statements specify the physical problems to be modeled which are structured on three levels: equations, models and parameters with the further distinction between a global or a local definition. Here we just explain the statements whereas the available physical models are discussed in Chapter 5 *Physical Models*.

With the option GlobalSpec the specification is global and holds over the whole simulation domain, whereas with the option MaterialSpec the specification is local and only holds on those regions of the simulation domain where the material has been defined. A material is associated within a region with the statement Material explained on p. 52. If the name NAME is not a defined material, it will be defined as a new material and initialized with default values or, if the option From is used, from the material mat0. The first defined material is the one mapped by default on the whole simulation domain.



SESES name	Description
<ClassCOORD>	All built-ins of ClassCOORD, see Table 3.6.
<Material Parameter>	The material parameters themselves, recursion is not permitted. See the Chapter 5 <i>Physical Models</i> and Table 5.3.
<Model Parameter>	A useful list of physical model parameters including primary- or dof-fields and secondary- or flux-fields. See the Chapter 5 <i>Physical Models</i> and Table 5.4.
<Element Field>	The element fields defined with the statement ElmtFieldDef with read-write access for ClassMODEL and read-only access for the subclass ClassMODEL_RO, see p. 49.
TangentX.<T1>, TangentY.<T1>, Normal.<T1>, DMapX.<T1>, DMapY.<T1>, DMapZ.<T1>,	The two tangent vectors and the outward normal to the boundary and the three tangent vectors of the isoparametric mapping. These two triads of vectors are defined whenever we either traverse parts of the boundary or of the domain but otherwise lose their meaning since they are aliased and have the same values.

Table 3.10: The built-in symbols of ClassMODEL\_RO and ClassMODEL available for defining material laws and material parameters. ClassMODEL\_RO is a subclass of ClassMODEL.

SESES name	Description
<ClassMODEL_RO>	All built-ins of ClassMODEL_RO, i.e. the class ClassMODEL but just with read-only access for the element fields, see Table 3.10.
<Boundary Field>	The boundary fields defined with the statement BoundFieldDef with read-write access for ClassBMODEL and read-only access for the subclass ClassBMODEL_RO, see p. 49.

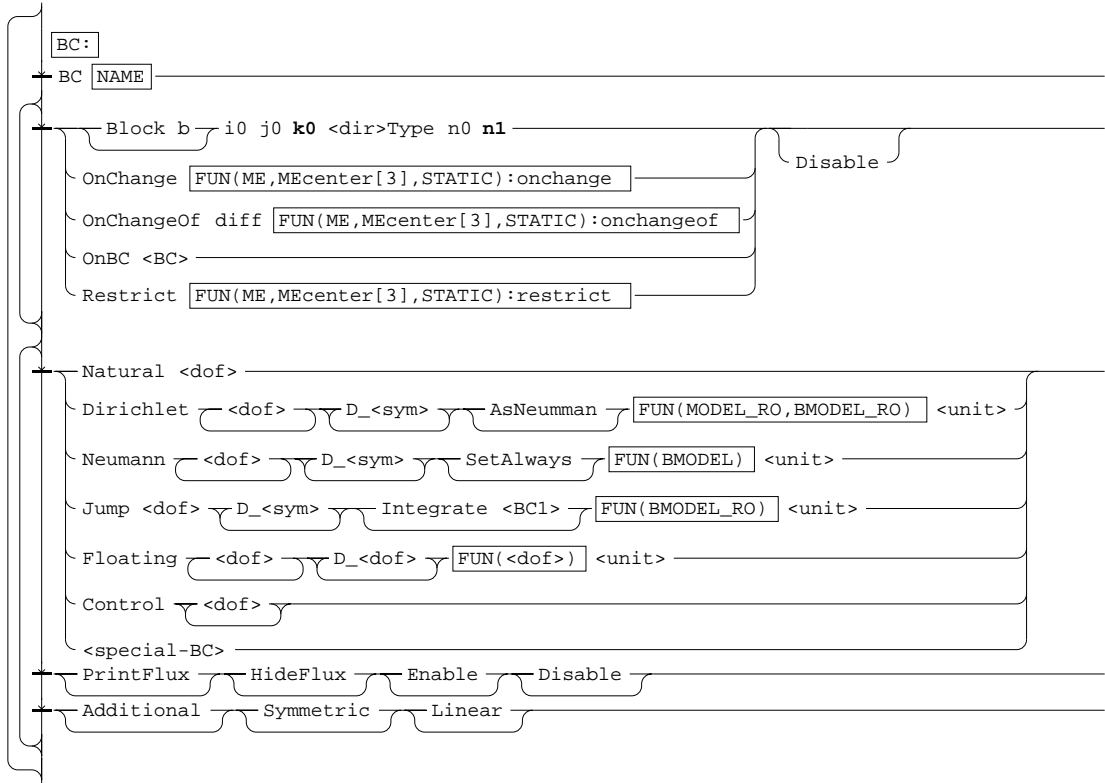
Table 3.11: The built-in symbols of ClassBMODEL\_RO and ClassBMODEL available for defining Neumann BC and Jump BC values. ClassBMODEL\_RO is a subclass of ClassBMODEL.

With the option `Equation`, only available together with the `MaterialSpec` option, one defines the governing equations to be solved for. In general, multiple equations can be solved locally and for each equation a number of dof-fields will be defined for which boundary conditions must be specified. The specification of boundary conditions is done with the statement `BC` explained on p. 57 and the list of equations and associated dof-fields is given in Table 5.1. Here, different equation names may be used to solve the same governing equation with different numerical methods. Since the dof-fields are only defined and computed where the associated equations are defined, in order to save on computational resources, one should define equations only on those regions of the simulation domain where they are of interest. This implies that for each different combination of equations, a different material must be defined on the simulation domain.

With the option `Model`, physical models are defined which tailor the governing equations to the specific needs of the simulation problem and allow a variety of similar physical systems to be studied. A default model is always defined when solving the governing equations, but other and more enhanced models can be selected with this statement. In general, non-default models are used to specify numerical methods or to turn on more complex models not available as default for efficiency reasons. If a model additionally depends on some other parameters, they must be specified either right away within parenthesis and/or with the `Parameter` option, discussed next.

In turns, equations and models can depend on many parameters which can be specified with the option `Parameter`; this option may also be omitted. These parameters which are listed in Table 5.2-5.3 are actually function definitions which are independent from the equation or model activation and they will be evaluated only if the associated equation or model has been enabled. An equation or model parameter can be a scalar, a vector, a tensor or some other similar quantity. If multiple values must be defined, one can specify all of them in a row of input values but it is also possible to singularly specify each component value by suffixing the parameter name with a dot followed by the component name. The first method requires to know the order in which multiple components must be specified and the second method requires to know the component names, see for example Table 3.1. The values of global parameters are always locally constant and so they can be accessed everywhere, see Table 3.3, but the values of material parameters `par(MODEL)` can also be functions of some or all the built-in symbols of the input class `ClassMODEL`, see Table 3.10. This input class defines symbols for accessing all primary- or dof-fields, the secondary- or flux-fields as well as other useful physical model parameters. This class also has read-write access to the element fields defined with the statement `ElmtFieldDef` explained on p. 49, but here the set of points used for evaluation and definition must agree, otherwise the Kernel program stops operation. If write access is not required, to avoid these inconsistencies, one can use the subclass `ClassMODELRO` with read-only access to the element fields but no other restrictions. A very similar class is `ClassBMODEL` used to evaluate boundary functions like Neumann BCs which has `ClassMODELRO` has a subclass but in addition has read-write access to the boundary fields defined with the statement `BoundFieldDef` explained on p. 49. Here again the set of points on the boundary used for evaluation and definition must agree otherwise the Kernel program stops operation and for read-only access, one can alternatively use the subclass `ClassMODELRO` with read-only access but no further restrictions. It is important to note that the built-in symbols are defined with respect to an internal unit which must be considered when constructing algebraic expressions for material parameters, see also the Section 3.4 *Classes of built-in parameters and functions*. The specification of locally non-constant material parameters may also include the specification of some derivatives used by the numerical algorithm and if defined, these derivatives with respect to some or all built-in symbols should always be correctly specified. For some material parameters and because of the large number of available derivatives, the derivatives of interest must be first declared with the option `D_<sym>` and the derivative's values must follow, in the same order of declaration, the parameter's value. Derivative's evaluation may be dynamically skipped if the built-in symbol `NODERIV` is non-zero, see Table 3.3. Changed parameters may be returned to default values with the option `Default`. The user can print the values of locally constant material parameters with the `Write MatSetting` statement described on p. 84.

## Initial-statement BC



With this statement boundary conditions (BC) for the dof-fields listed in Table 5.1 are specified. When solving partial differential equations, BCs are an integral part of the specification required to solve the governing equations for the dof-fields and they represent the interaction with the external world. Basically BCs define the dof-fields or some of their derivatives on portions of the domain boundary. For a brief presentation of the subject see the Chapter 4 *Numerical Models*.

A BC definition starts by giving a name `NAME` used for later reference and it is followed by the definition of the boundary as a subset of the simulation domain boundary or a surface on its interior. This boundary is composed by one or more macro element corners, edges or faces. A face can be selected by first defining a macro element node with the block indices `i0 j0 k0` followed by the choice of two block directions `<dir>Type` e.g. `IJType` and two segment lengths `n0 n1` along the selected directions. The available block directions are given in Fig. 3.1 and depend on the block macro element shape. The macro element block can be selected with the `Block` option and block number `b`, otherwise the last selected block is used. By using the zero lengths, the macro element faces can be contracted to a line of macro element edges or a single macro element node. The option `OnChange` selects as boundary surface those macro element faces when the function `onchange(ME,MEcenter)` evaluated on both sides of the face returns a zero and a non-zero value. In addition to the built-in symbols of the input class `ClassME`, see Table 3.5, this function has access to the built-in `MEcenter` representing the ME center. An undefined or removed macro element corresponds to a zero value and thus the constant function 1 selects the whole boundary. The option `OnChangeOf` works similarly but the boundary is selected if

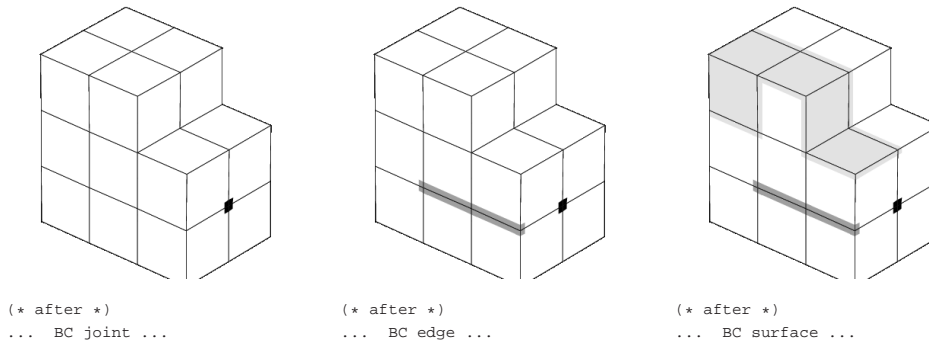


Figure 3.2: Example of a joint, an edge and a surface boundary.

the difference of both evaluations of the function `onchangeof(ME,MEcenter)` is  $\pm \text{diff}$ . The option `OnBC` adds a previously defined boundary. With the `Disable` option, the specified part of the BC surface is deleted. The `Restrict` option is used to additionally restrict the macro elements belonging to the boundary which per default are all ones touching in some way the boundary. Selected are only those macro elements where the function `restrict(ME,MEcenter)` is non zero. Do not use this option just as an additional way to restrict the boundary to a region, it is for expert users only. The definition of the boundary is done just once when parsing and therefore any dependency of the maps `onchange`, `onchangeof` and `restrict` from user parameters, as explained in Section 3.2 *User Parameters*, is not considered anymore after parsing.

Here a simple example of a BC boundary definition illustrated by Fig. 3.2.

```
QMEI 3 1 QMEJ 2 1 QMEK 3 1
DeleteME block(2 3 0 2 0 1)
BC joint    3 1 2 JKType 0 0
BC edge     1 2 2 IJType 2 0
BC surface  0 1 0 IJType 2 1  2 1 0 JKType 1 1
            2 1 1 IJType 1 1  0 2 0 IKType 1 1
```

After the definition of the boundary, one defines the numerical BC consisting of a BC type, a list of dof-fields `<dof>` for which the BC applies and the BC values which may depend on the built-in symbols of the associated input class. In general, each specified `<dof>` may be a scalar field, a vector field or the component of a vector field, but some restriction applies for different cases. For non-constant BC values and if the system allows the user to specify their derivatives with respect to some or all built-in symbols, then the user should correctly specify all these derivatives. Derivative's evaluation may be dynamically skipped if the built-in symbol `NODERIV` is non-zero, see Table 3.3. BCs can be defined for any system defined dof-field, however, they will be used only if governing equations for the dof-fields are being solved. The following BC types are available.

- For the `Natural` BC type, no boundary conditions are set. However, for this BC as with any other type of BC, the Kernel program evaluates and displays the BC flux through the BC surface.
- For the `Dirichlet` BC type and on the BC surface, the dof-field value of `<dof>` is set to the given function of the input class `ClassMODELRO`, see Table 3.10. If the

Dirichlet value depends on symbols of Table 3.10 which are dof-fields dependent, then you should correctly specify all derivatives with the option `D_<sym>` and the derivatives must follow, in the same order of declaration, the Dirichlet value. In this case one is actually defining constraint equations for the dof-field `<dof>`. If multiple dof-fields are defined, just define the multiple values in a row. With the option `AsNeumann`, the evaluation is done as for Neumann BCs on the BC surface of positive measure and with a BC value of the input class `ClassBMODELRO`.

- For the Neumann BC type and on the BC surface with a positive measure, the normal flux component of `<dof>` is set to the given function of the input class `ClassBMODEL`, see Table 3.11. For a vector dof-field all components must be specified and so use null values for unused components. If some of the BC surface lies on the domain interior and therefore it is not part of the domain boundary, the Neumann BC is set on both sides and the result is a discontinuous normal flux with discontinuity twice the Neumann value. To set the BC on one side only, you may use the `Restrict` option or a BC value differentiating between both sides. Per default, the Neumann BC will be set only if the dof-field is locally defined within the elements touching the boundary, however, with the `SetAlways` option is enough for the dof-field to be defined over the boundary i.e. is enough for the dof-field to be defined on the other side of the boundary. If the Neumann value depends on symbols of Table 3.11 which are dof-fields dependent, then you should correctly specify all derivatives with the option `D_<sym>` and the derivatives must follow, in the same order of declaration, the Neumann value. If multiple dof-fields are defined, just define the multiple values in a row.
- For the Jump BC type and on the BC surface with a positive measure, the dof-field `<dof>` will be a double valued function, with the discontinuity between both BC sides set to the given function of the input class `ClassBMODEL`, see Table 3.11. You must use the `Restrict` option to specify on which side of the boundary the discontinuity has to be applied, otherwise it will set on both sides resulting in a useless shift of the dof-field at the boundary. The jump value represents the dof-field discontinuity for going from the selected boundary's side to the other side. The `Integrate` option is used to evaluate the jump's value as a boundary integral on `<BC1>`. If the jump value depends on symbols of Table 3.11 which are dof-fields dependent, then you should correctly specify all derivatives with the option `D_<sym>` and the derivatives must follow, in the same order of declaration, the jump value.
- For the Floating BC type, the normal component to the boundary of the flux for the dof-field `<dof>` integrated over the BC surface is prescribed to the given value and the dof-field value over the boundary is set to be a constant first determined by the solution algorithm in order to satisfy the amount of prescribed flux. This flux value can itself be a function of the constant dof-field values `<dof>` associated with this BC and if this is the case, one should correctly specify all derivatives with the option `D_<dof>` and the derivatives must follow, in the same order of declaration, the flux value. If multiple dof-fields are defined, just define the multiple values in a row.
- The Control BC type is used to change the `Enable`, `Disable`, `HideFlux` or `PrintFlux` flag values for already defined numerical BC.

- The special-BC BC type is a convenient model-related form converted into one or a combination of the above BC types. These special BCs are discussed in the Chapter 5 *Physical Models*.
- Periodic BCs are also available in *SESES*, however, since they cannot be defined as freely as the other BCs, they are defined indirectly using the `JoinME` statement described on p. 53.

After the definition of the numerical BC, some additional flags may be specified. Per default, defined BCs are active and BC fluxes are displayed on the output after a solution. However, the options `Enable`, `Disable`, `HideFlux` and `PrintFlux` are available to change this default behavior. In general, if for a dof-field several BCs are defined with overlapping or touching surfaces, most of the time unpredictable results are obtained. Therefore per default, only one numerical BC is active for a given BC and dof-field which is always the last defined one. In the seldom occasion where several numerical BCs should be contemporary active, as for example with a Neumann and a jump BC, then the `Additional` option must be given to keep all previously defined values. In general, for BCs with derivatives the contributions to the solver are generally non-symmetric and non-linear, however if they are symmetric or linear, the options `Symmetric` and `Linear` help *SESES* selecting the correct solver.

For example, the solution of this simple 2D Poisson problem with two Dirichlet BCs for the dof-field `Phi`

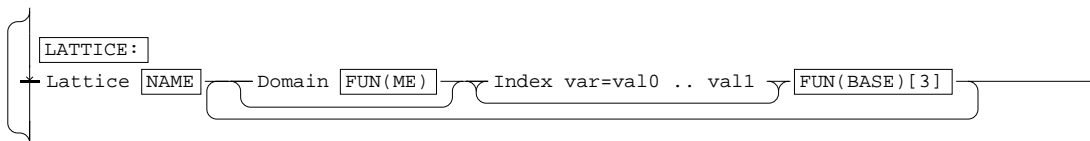
```
MaterialSpec Silicon Equation ElectroStatic
QMEI 2 1 QMEJ 2 1
BC Left IType 0 2 0 Dirichlet Phi 1 V
BC Right IType 0 2 2 Dirichlet Phi 0 V
Finish
```

produces at runtime the following output

```
<BC Name=Left>
  <Field= Phi Flux= Charge TypeBC= Dirichlet > 1.05364742e-10 C
<BC Name=Right>
  <Field= Phi Flux= Charge TypeBC= Dirichlet > -1.05364742e-10 C
```

giving the total electric charge at the capacitor's contacts.

## Initial-statement `Lattice`



This statement defines a *SESES* lattice, i.e. a list of grid points placed anywhere in the simulation domain. Thereafter and with the `Write` statement of the `Command` section explained on p. 84, the user can print field values on the lattice points. A lattice name `NAME` must first be defined for later reference. Afterwards, from zero up to three different index variables `var` with integer values from `val0` to `val1` can be



ReadMesh(Fortran; Coord <span>LITERAL</span> :C; ShapeQ <span>LITERAL</span> :SQ; ShapeT <span>LITERAL</span> :ST; ShapeP <span>LITERAL</span> :SP; <span>LITERAL</span> :filename )
Read an unstructured ME mesh from the file filename in a line-oriented coordinates and incidences format. By choosing any numbering for the ME nodes, for each ME node one writes a line prefixed by the string C, followed by the ME node number and the ME node coordinates. Afterwards for each ME, one writes a line with the prefixes SQ, ST or SP for a hexahedron, tetrahedron or prismatic ME shape, followed by the ME number and the ME node numbers in the standard and commonly used FE node order. With the option Fortran, the node numbering must start by one otherwise by zero. The prefixes can be changed with the options Coord, ShapeQ, ShapeT and ShapeP. The file's contents is not preprocessed except for lines starting with a # character treated as a comment line and discarded. Each specified ME results in a single ME block with ME joins which can be further processed as any other defined ME.

Table 3.12: Miscellaneous initial task built-in functions.

defined to be used for the definition of the lattice points thus forming a 0-, 1-, 2-, or 3-dimensional basic lattice. Note that the two dots ellipses . . between the lower and upper index value is part of the syntax and must be present. Multiple basic lattices can be combined together to form a single lattice. Use the built-in function node to obtain the coordinates of macro element nodes. The option Domain just considers lattice points where the domain function is not zero.

In the next example, a one-dimensional lattice is defined with 11 equally distributed points between the (0, 0, 0)- and the (1, 1, 1)-node of the first macro element.

```
Lattice lat Index i=0..10 node(0,0,0)+(node(1,1,1)-node(0,0,0))*i/10
```

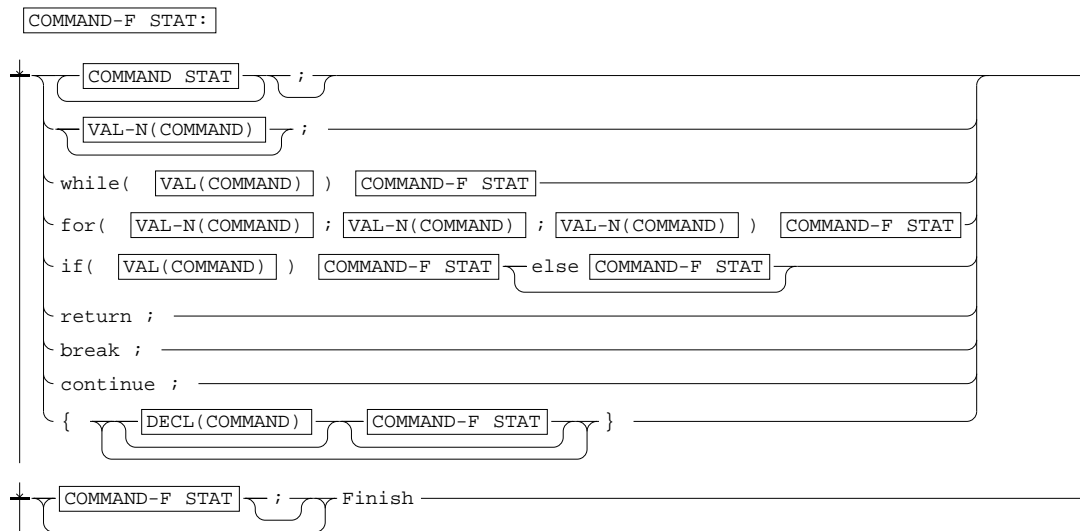
## Initial-statement Misc



This statement provides routines for replacing a collection of input statements. In general, these are convenience routines and do not provide more capabilities.



### 3.11 Syntax of the command section



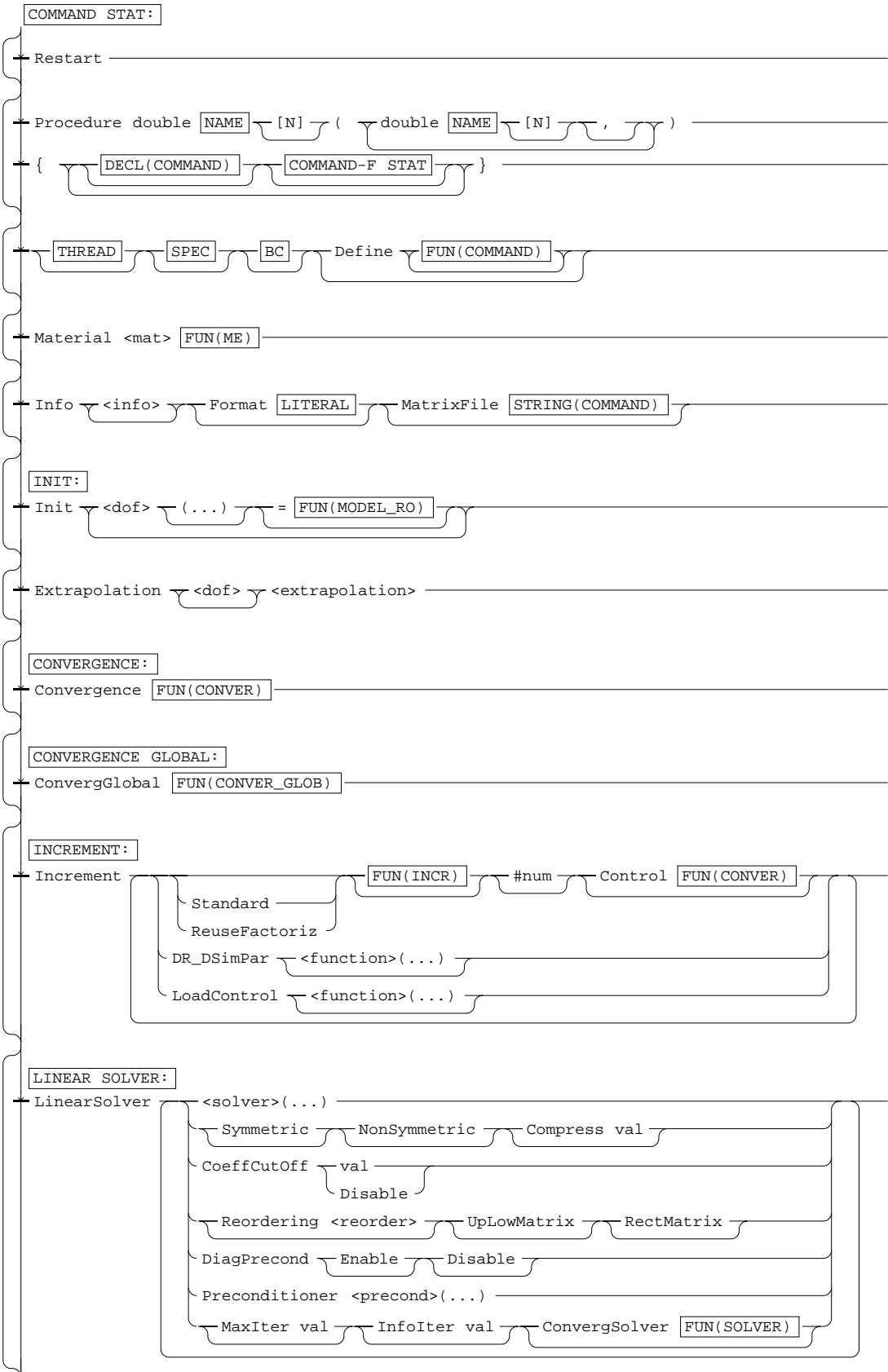
Similarly to the initial section, the command section consists of a series of statements of type `COMMAND STAT` or `COMMAND-F STAT` in any arbitrary order and terminated by the `Finish` symbol. However, differently from the initial section which has a static approach, we have here a more flexible dynamical approach with the plain command statements of type `COMMAND STAT` embedded in the extension of a functional block called `COMMAND-F STAT`, see the Section 3.5 *Block Functions*. The plain command statements of type `COMMAND STAT` which are not separated by a semicolon are grouped together to form a single statement which need to be considered when using conditional control statements without limiting braces. The semicolon may also be necessary in order to stop reading the actual statement, if the following part may belong both to the actual and next statement.

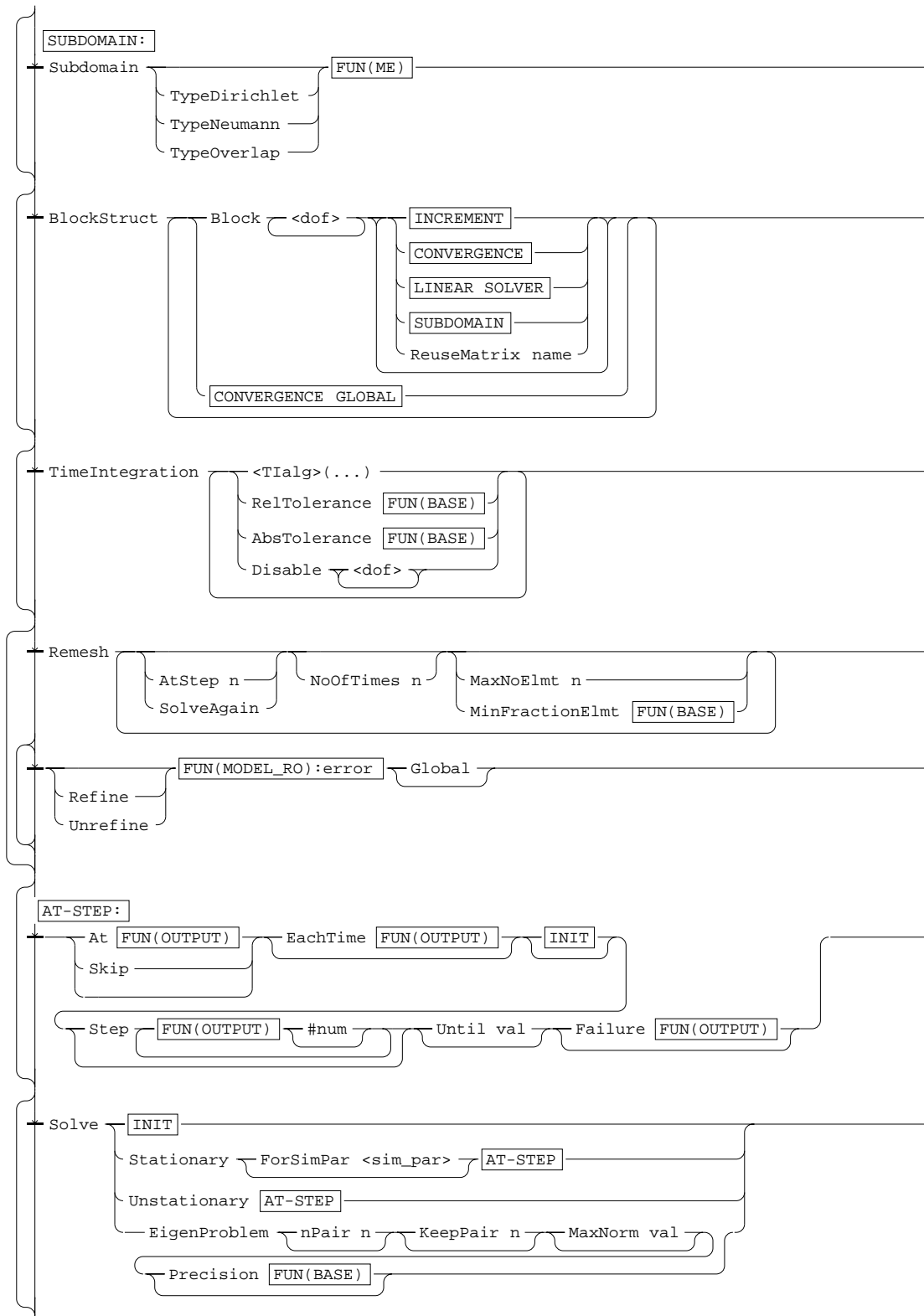
The numerical values of the command statement `COMMAND-F STAT` belong to the input class `ClassCOMMAND` composed of the input class `ClassOUTPUT`, see Table 3.13 together with the user command procedures defined by the statement Procedure as explained on p. 67.

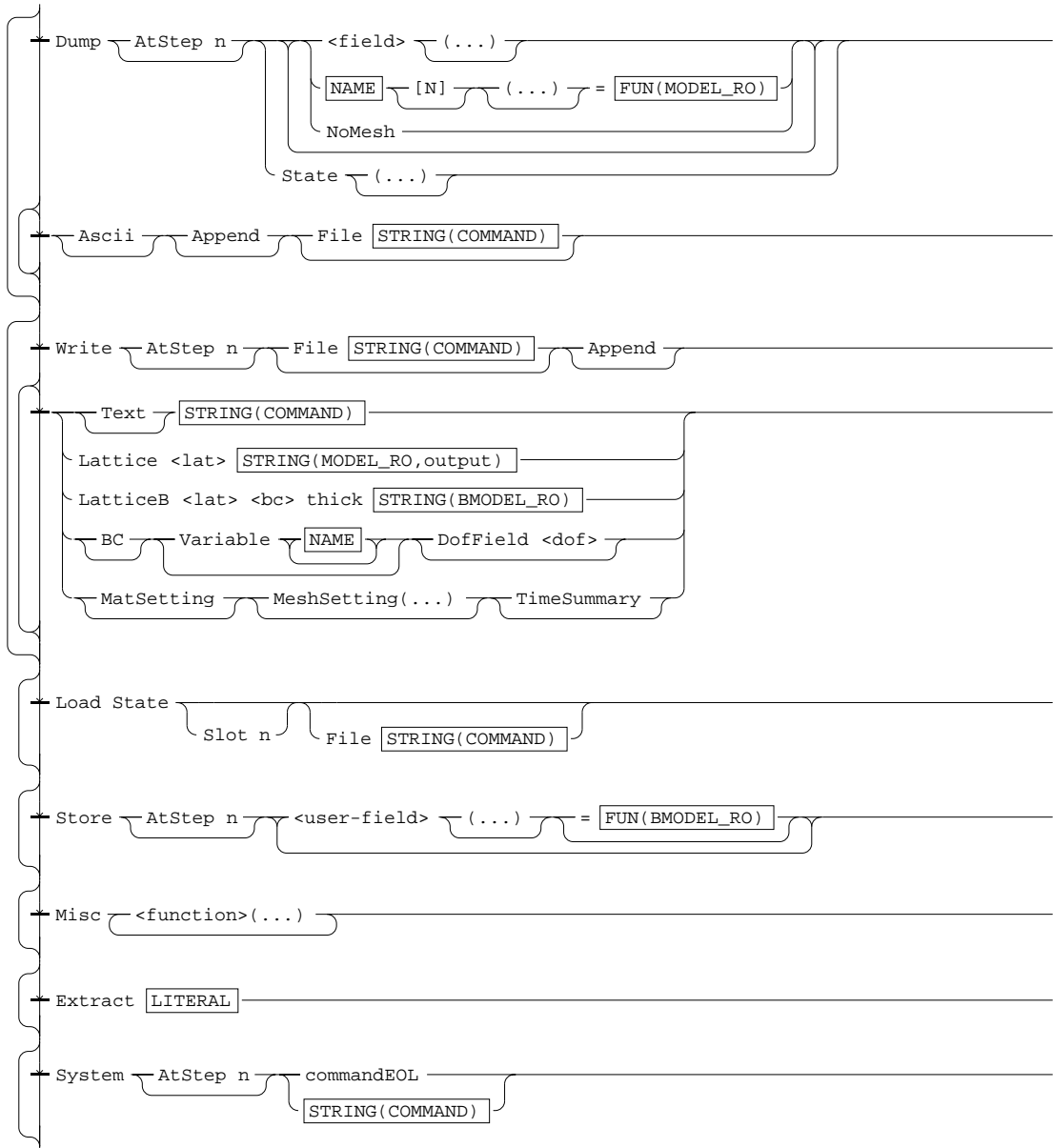
<code>&lt;BC&gt;.&lt;Field&gt;.Flux, &lt;BC&gt;.&lt;Field&gt;.Shift</code>
These built-ins characterize BCs. For the last computed solution, the first form gives for the BC <code>BC</code> and the dof-field <code>&lt;field&gt;</code> the integrated flux through the BC surface. The second form gives for a BC of type <code>Floating</code> the computed shift.
<code>eigenvalue.n&lt;num&gt;, eigenvalue, eigennum</code>
These built-ins access the eigenvalues of a computed eigen solution sorted in increasing order. The first form access the <code>num</code> -eigen value and the second form the eigenvalue of the eigenvector stored in memory whose number is <code>eigennum</code> . At the end of a eigen solution, it is only possible to keep in memory one eigenvector selected with the <code>KeepPair</code> option of the <code>Solve Eigen</code> statement. However, with the <code>AtStep</code> option of the <code>Write</code> and <code>Dump</code> statements it is possible to access each eigen-pair since they are all ones in turn stored in memory by updating the values of <code>eigenval</code> and <code>eigennum</code> .
<code>double output[n](fun(MODEL_RO)[n]; at(OUTPUT).&lt;T1&gt;)</code>

<p>This built-in returns the evaluation of the function <code>fun(MODEL_RO)</code> at the point <code>at</code>. It is the functional form for defining a one point lattice with the statement <code>Lattice</code> and evaluating a function at the single lattice point with the <code>Write Lattice</code> statement.</p>
<pre>double integrate[n](Domain dom(ME); Bound [&lt;bc&gt;]+; Lattice [&lt;lat&gt;]+; Type val; fun(BMODEL_RO)[n])</pre>
<p>This built-in integrates the <math>n</math>-dimensional function <code>fun(BMODEL_RO)</code> of class <code>ClassBMODEL_RO</code> over the domain where <code>dom(ME)</code> is not zero, the boundaries <code>[&lt;bc&gt;]+</code> or the lattices <code>[&lt;lat&gt;]+</code>. The notation <code>[ ]+</code> represents one or more items and if no option is given the integration is over the whole domain. Without the option <code>Type</code> or with <code>Type 0</code>, two Gauss quadrature points per dimension and element are used whereas <code>Type 1</code> allows approximate integration with three points. In 3D, a domain integral is over a volume, a boundary integral is over a surface and for a lattice the integration's type depends on the lattice definition. For a 0-dimensional lattice the integrand at the single lattice point is added to the final result without any weighting and for a 1-, 2- or 3-dimensional lattice, a line, surface or volume integral is performed as defined by the lattice points. The built-in <code>Normal.&lt;T1&gt;</code> is available to define the integrand function <code>fun</code> and represents the outward normal for boundaries, the surface normal for 2-dimensional lattices, the line tangent for 1-dimensional lattices and the zero vector otherwise.</p>
<pre>double maxvalue[n](Domain dom(ME); Bound [&lt;bc&gt;]+; Lattice [&lt;lat&gt;]+; Type val; Nodal; fun(BMODEL_RO)[n])</pre>
<p>This built-in is similar to <code>integrate</code> but returns the absolute maximal value of <code>fun(BMODEL_RO)</code>. With the option <code>Nodal</code>, the integrand is evaluated at Lagrangian nodal points instead of quadrature points.</p>
<pre>double traverse[n](Domain dom(ME); Bound [&lt;bc&gt;]+; Lattice [&lt;lat&gt;]+; Type val; Nodal; fun(BMODEL_RO)[n])</pre>
<p>This built-in is similar to <code>maxvalue</code> but just returns the sum of <code>fun(BMODEL_RO)</code> at the evaluation points.</p>
<pre>double residual[n/DIM](Bound [&lt;bc&gt;]+; Type val; fun(BMODEL_RO)[n])</pre>
<p>This built-in integrates the <math>n</math>-dimensional function <code>fun(BMODEL_RO)</code> of class <code>ClassBMODEL_RO</code> over the boundary <code>[&lt;bc&gt;]+</code> using the residual approach, see also the Section 4.2 <i>Current Conservation</i>. Let assume we have a solenoidal vector field <math>\mathbf{F}</math> with <math>\nabla \cdot \mathbf{F} = 0</math>, then the surface integral on any domain <math>\Omega</math> is zero <math>\int_{\partial\Omega} \mathbf{F} \cdot d\mathbf{n} = 0</math>. For the discretized equations, this relation may not be true, since <math>\nabla \cdot \mathbf{F} = 0</math> may hold just in a weak sense and a better zero balance may be obtained by the following argument. We introduce the weight functions <math>H_j</math> with <math>\sum_j H_j = 1</math> and <math>H_j(\mathbf{x}_i) = \delta_{ij}</math> where <math>\mathbf{x}_i</math> are the mesh nodes. Practically, these weight functions are taken to be the nodal shape functions used when solving for the potential associated with <math>\mathbf{F}</math>. From the property <math>\sum_j H_j = 1</math> and <math>\int_{\partial\Omega} \mathbf{F} H_i \cdot d\mathbf{n} = 0</math> for any internal node <math>\mathbf{x}_i</math> not on <math>\partial\Omega</math>, we obtain the relation</p> $\int_{\partial\Omega} \mathbf{F} \cdot d\mathbf{n} = - \sum_{\forall \text{Node } j \text{ on } \partial\Omega} \int_{\Omega} \mathbf{F} \cdot \nabla H_j d\Omega = 0.$ <p>We have now converted the surface integral into a volume integral and just the contributions from nodes on the boundary are non-zero and therefore these nodes can be easily identified with a given boundary. In particular, we see that elements having just a point in common with the boundary have in general non-zero contributions.</p>
<pre>double store(...; &lt;user-field&gt;[=fun(MODEL_RO)[n]])</pre>
<p>This built-in is the functional form of the <code>Store</code> statement explained on p. 86 and has the same options ... of Table 3.26.</p>

Table 3.13: Built-in symbols of class `ClassOUTPUT` for output and post-processing.





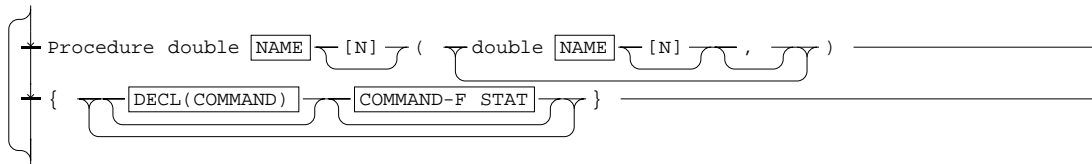


## Command-statement Restart



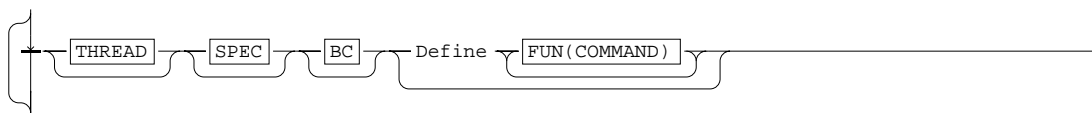
The statement `Restart` performs a full restart of the Kernel program. Within the same process, a new pristine system is created, the control variable `RESTART` is incremented by one and the input is newly read. The initial value of `RESTART` is one for the Kernel program and it is always zero for the Front End graphical program.

## Command-statement Procedure



This statement defines command procedures which are an extension to user defined routines, see Section 3.6 *User Routines*. These command procedures together with the input class `ClassOUTPUT`, see Table 3.13, define the input class `ClassCOMMAND`. Command statements of type `COMMAND-F STAT` are collected together into a procedure named after `NAME` and to be called as a routine of the input class `ClassCOMMAND`. The definition of procedures can be nested, but procedures can only be called within the actual block of command statements. The procedure parameters and the procedure variables defined by the syntax box `DECL` are accessible as global variables, however, if the procedure is not active they will have zero values. This is indeed possible since several command statements may just register functional calls for later evaluation even outside the procedure call.

## Command-statement `THREAD` `SPEC` `BC` Define



These statements are the same as the ones of the `Initial` section explained on p. 47, 54, 57, 47 and they allow to dynamically change these properties at run-time. The only limitation is that with `BC` it is not possible to change BC boundaries. With the `Define` statement, user parameters are changed but differently from the initial section, the input class is now `ClassCOMMAND` and the evaluation order among several `Define` statements can be generic and is given by the command execution's flow. If a new material is defined with `SPEC` statement, the material will be either initialized with default data or, if the `From` option is used, with the associated material data as found at the end of the `Initial` section. Further, all changes of equations and models cannot be viewed within the graphics Front End program and changes in parameters can only be viewed by writing graphics data files. If equation properties are changed, as far as possible, we try to keep all actual numerical data except for data related to time integration algorithms.

## Command-statement Material



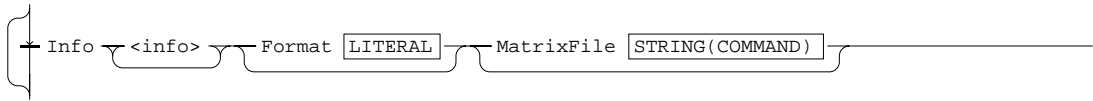
This statement is the same as for the initial-statement discussed on p. 52. At the present time, it is not possible to visualize within the graphics Front End program the

<info>	Description
HideBC	The BC info at the end of each solution step is not displayed.
TimeIteration	The start of a time or parameter iteration is displayed.
Iteration	The start of a global and block iteration is displayed.
LinearSystem	Some informations on the linear solver are displayed.
Refine	Additional informations on refinement.
LinearMatrix	The matrix, right-hand side and solution vector are displayed at each linear step in a sparse format used by the program <i>Maple</i> .
LinearMatrixAllDof	The same of LinearMatrix but with the inclusion of Dirichlet dofs which have a zero solution.

Table 3.14: List of &lt;info&gt; choices.

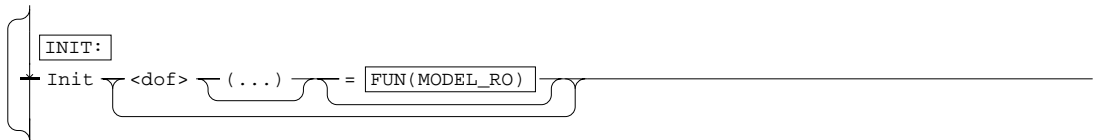
new material mapping and each time this statement is performed all actual numerical data is lost.

### Command-statement Info



This statement provides the user with the information <info> on the output stream during execution of the Kernel program. The list of the most relevant <info> choices is given in Table 3.14. Not displayed are items intended for debugging purposes that may not be available since special compilation options must be used. The option *Format* is used to specify the print format of numerical values and must conform to the ANSI C standard for the `printf` family of routines and double values. The option *MatrixFile* writes to the specified file instead of the standard output, the linear system matrix requested with the info *LinearMatrix* or *LinearMatrixAllDof*. Differently from the *Write* statement explained on p. 84, the data is not supplied immediately to the user but only during the execution of the Kernel program. A *Info* statement always replaces a previous one.

### Command-statement Init



This statement allows the default zero initialization of the dof-field <dof> to be replaced by the defined function. This initialization is performed when the statement *Solve Init* is called as described on p. 80 but does not apply at Dirichlet dof-values. If the function is not specified, the dof-field is left unchanged instead of being initialized and some optional parameters (...) are available to modify the default initialization. In general, this initial setting is only deterministic if the specified function is continuous, otherwise the initial value may depend on the way we traverse the mesh. Final computed solutions should not depend on the initial solution, however, to avoid



<extrapolation>	Description
NoExtrapolation	The solution vector of the previously computed solution is used.
Linear, Quadratic, Cubic, Quartic, Quintic	Polynomial extrapolation of degree 1, 2, 3, 4, 5.
LogLinear, LogQuadratic, LogCubic, LogQuartic, LogQuintic	Polynomial extrapolation of degree 1, 2, 3, 4, 5 of logarithmic values.
Rat_0_1, Rat_1_1, Rat_1_2, Rat_2_2, Rat_2_3	Rational extrapolation of degree 0+1, 1+1, 1+2, 2+2, 2+3.

Table 3.15: List of extrapolation functions.

the indeterminacy of a discontinuous function, one can use the option `Continuous` to average all contributions to a dof-value. The dof-field initialization order is the same one of declaration except for constant values which are set first. So it is possible to define values depending on already initialized dof-fields.

### Command-statement Extrapolation

Extrapolation {<dof> <extrapolation>}

This statement enables the initialization of solutions by extrapolation from previously computed solutions. The independent variable used for the extrapolation process is either the simulation time or a user parameter which is specified when computing solutions with the statement `Solve Stationary` or `Solve Unstationary` as described on p. 80.

The extrapolation procedure <extrapolation> is selected for the specified list of dof-fields <dof> or for all dof-fields if no dof-field is given. The list of available extrapolation procedures is given in Table 3.15.

When computing non-linear solutions which are smooth functions of the user parameter, an extrapolated solution can be a very good guess for starting the non-linear solution algorithm, leading to significant savings in iterations and computing time, see also the Section 4.3 *Non-Linear Solution Algorithms* for more details.

If for a field a high degree extrapolation is selected e.g. `Quadratic`, the Kernel program allocates memory to store three extra solution vectors. After two solutions have been computed, a linear extrapolation is automatically selected and after three solution steps, the `Quadratic` extrapolation can be performed.

### Command-statement Convergence

CONVERGENCE :  
Convergence FUN(CONVER)

Built-ins of ClassCONVER	Description
<ClassOUTPUT>	All built-ins of ClassOUTPUT, see Table 3.13.
nIter	Number of iterations.
RelIncr.<dof>	$L^2$ -norm of the <dof>-increment relative to the solution.
AbsIncr.<dof>	$L^2$ -norm of the <dof>-increment.
AbsResid.<dof>	$L^2$ -norm of the <dof>-residual
MaxIncr.<dof>	$L^\infty$ -norm of the <dof>-increment.
MaxResid.<dof>	$L^\infty$ -norm of the <dof>-residual.
prev(val)	The value of val(CONVERG) evaluated with norm values from the previous iteration.
prev2(val)	Shortcut for prev(prev(val)).
prev3(val)	Shortcut for prev(prev2(val)).
quot(val)	The quotient val/prev(val).

Table 3.16: Built-in symbols of ClassCONVER available when defining functions for convergence.

This statement defines the default convergence criterion of the Newton, coupled or block iteration loop when solving non-linear partial differential equations, see also the Section 4.3 *Non-Linear Solution Algorithms* for more details. The convergence criterion is a function of the input class ClassCONVER, see Table 3.16. At the end of each block iteration step, the criterion is evaluated and if non-zero, the block iteration is considered as converged and terminated. If the criterion is not specified, a default one applies as follows. If the block defines linear equations, a single block iteration is performed, otherwise we use the convergence criterion given by the expression `AbsResid.dof0<1E-6 && AbsResid.dof1<1E-6 && ...` with `dof0, ...` the dof-fields in the block.

Typically the convergence criterion is defined as a block function and used to additionally print messages or to stop the block iteration whenever divergence occurs by calling the functions `write` and `failure` defined by Table 3.3. Failure criteria are very useful together with automatic step algorithms meaning that the chosen step was too large and a new solution should be tried with a shorter step. If the convergence criterion is not specified, the default failure criterion reads `nIter>=15`.

## Command-statement ConverGlobal



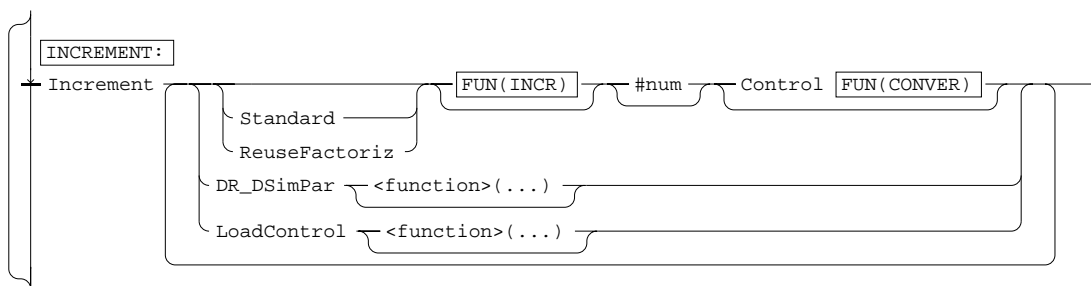
This statement is equivalent to the previous one, but it defines the default convergence behavior of the Gauss-Seidel, uncoupled or global iteration loop. This iteration is only defined if more than one block is defined. The convergence criterion is similarly defined but the values of the built-in symbols listed in Table 3.16 refers now to the norms computed in the first iteration of each block. If different block defines the same dof-field, the norm value for that dof-field will be the last computed one. The default convergence criterion reads `nIter>=1` i.e. a single global iteration step is performed. This behavior leads to a correct computation of solutions only if there

<ClassCONVER>
All symbols of class ClassCONVER, see Table 3.16.
double setincr(DofField [<dof>]++; fun(CONVER,Incr,Dof))
For each single block-dof, this function sets the increment according to fun. This value belongs to the input class ClassCONVER but has additionally access to the computed negative increment with Incr and actual solution with Dof before increments subtraction takes place. With the option DofField one can limit the action to selected block dof-fields [<dof>]++. The function setincr always returns zero.
double setnorm([ AbsResid.<dof>   AbsResid.<dof>   MaxResid.<dof> ]+)
This function updates the value of the given residual norms. Per default, their values is the one computed before solving the block linear system. The function setnorm always returns zero.
double setdof(Domain dom(ME); <off-block-dof>; fun(CONVER,Incr.<dof>,Dof.<dof>))
This routine is similar to setincr but it is used to amend off-block dof-fields and sets the solution of the dof-field <off-block-dof> according to fun. This value belongs to the input class ClassCONVER but has additionally access to any dof-field negative increment with Incr.<dof> and dof-field solution Dof.<dof> before increments subtraction takes place. Undefined block dof-fields have zero increments. With the option Domain, one can limit the setting to the selected domain. The function setdof always returns zero. One should avoid calling this function to amend block dof-fields, if possible use setincr, which runs much faster.

Table 3.17: Built-in symbols of ClassINCR available when defining functions for increment amendments.

is no coupling between the different blocks or if the coupling is strictly unidirectional starting from the first to the last defined block. Clearly this is not the general case, but many problems display this unidirectional coupling scenario and so the default behavior will work correctly for this class of problems. The default printing behavior is to print nothing at the end of each global iteration step.

## Command-statement Increment



This statement defines the default behavior for computing and setting block dof-field increments before they are added to the solution. Block dof-field increments are computed for all dof-fields defined inside the block structure at each linear solution step. As an example, for non-linear solutions, the increments computed by Newton's algorithm may be too large, so that to improve convergence they are generally *shortened* during the first iterations. Many other algorithms like line search methods may use

Numerical( <i>eps</i> )
This function computes the derivative by numerical difference, it is a very general method but due to numerical cancellation it may be unstable. The derivative of the residuals $\mathbf{R}(\lambda)$ with respect to the user parameter $\lambda$ is computed by the difference $\partial\mathbf{R}(\lambda)/\partial\lambda = (\mathbf{R}(\lambda + \Delta\lambda) - \mathbf{R}(\lambda))/(\Delta\lambda)$ with $\Delta\lambda = \text{eps}$ . The parameter <i>eps</i> must be chosen carefully.
Linear(BC [ <i>bc0.bc1</i> ]+; Material [ <i>mat0.mat1</i> ]+; NoRhs; NoLhs)
This function computes an exact derivative but requires the linear dependency of the governing equations from the terms depending on the user parameter. However, linearity with respect to the user parameter is not required. Neumann BCs, floating BCs and many material parameters are linear within the governing equation therefore the derivative can be easily computed by substituting the original value with its derivative and performing a normal assembling step. For this purpose one has to define new BCs and materials which per default are disabled and where the parameter values are the derivatives with respect to the user parameter. The disabled BCs representing the derivative of the active BCs are defined with the option BC. The disabled BC <i>bc1</i> will substitute the active BC <i>bc0</i> . The disabled material with material parameters representing the derivative of material parameters are defined with the option Material. The disabled material <i>mat1</i> will substitute the active material <i>mat0</i> . The equation setting of <i>mat1</i> is not used, however, everything not depending on the user parameter must be zeroed in <i>mat1</i> . The notation [ ] represents an optional item and [ ]+ one or more items. The options NoRhs, NoLhs may be used to speed-up the computation meaning that the left-hand-side or right-hand-side of the governing equations do not depend from the user parameters.

Table 3.18: Methods to compute the derivative of the residual equations with respect to user parameters.

this paradigm and some are presented in the Section [4.3 Non-Linear Solution Algorithms](#).

The actual computation of the block increments is determined by the options `Standard` and `ReuseFactoriz` where the first one is the default one. With the former choice, the increments are computed by solving the block linear system, i.e. a standard full linear step is performed, whereas with the latter one, one tries to reuse the factorized linear matrix of the previous linear step. Afterwards with a function of the input class `ClassINCR`, see Table [3.17](#), there is the possibility to modify the computed increments before they are added to the solution. With the option `#`, this specified increment directive is repeated *num* times, otherwise just once and then one passes to the next defined one. When all increment directives with their repetitions have been evaluated, the list is started again. To break down this periodic cycle, the option `Control` may be used to jump ahead ( $> 0$ ) or back ( $< 0$ ) at the actual list position where the jump value is a function of the input class `ClassCONVER`, see Table [3.16](#). If the `Control` option is used the default value of the repetition *num* is  $\infty$  otherwise one.

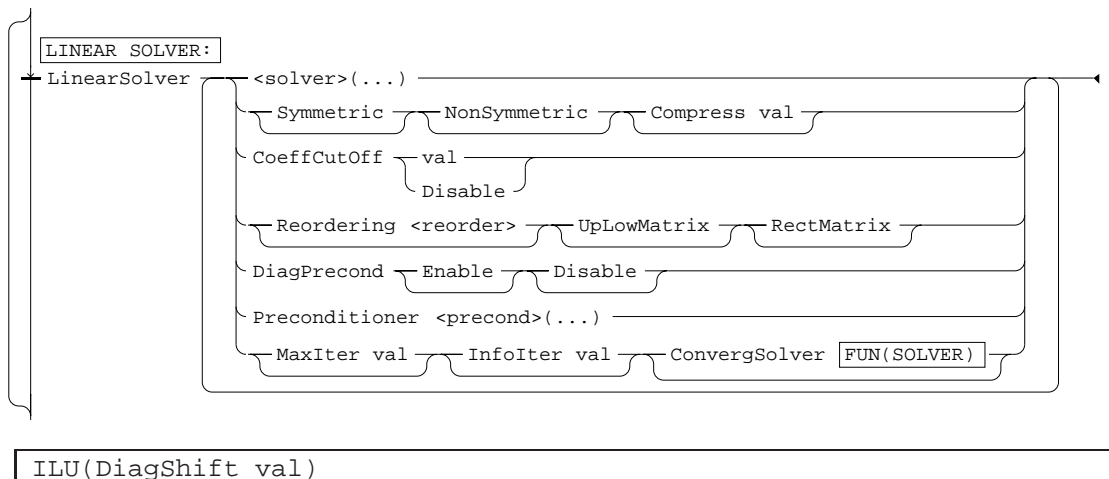
When computing families of solutions, special numerical algorithms computing the increments may require the derivative of the residual equations with respect to changes of the user parameter. This derivative is specified with option `DR_DSImPar` and the following functions. The choices always available are documented in Table [3.18](#), but other special purpose functions may be added by the Kernel program whenever special model dependent code is required to compute the derivatives otherwise not available within the default functions.

<code>Spherical(Predic val(step); FacIncr fu; FacLoad fl; sim_par)</code>
This function uses the spherical arc-length parametric equation (4.33) together with the exact form (4.34) to stabilize the continuation method with respect to the load parameter $\lambda = \text{sim\_par}$ . Each time a new solution is started, the value of $\lambda$ is predicted as $\lambda + \Delta\lambda_0$ , where $\Delta\lambda_0$ is defined by the option <code>Predic</code> as a function of the built-in <code>step</code> representing $\Delta s$ . Its default value is $\Delta\lambda_0 = \Delta s / \text{fl}$ if $\text{fl} \neq 0$ otherwise $\Delta\lambda_0 = \Delta s$ . The value $\Delta\lambda_0$ is also used to give the direction for the first solution step and thus to select one of the two solutions for the update $\Delta\lambda$ . For all other steps, the direction will be the one most close in angle to the previous direction. The <code>FacIncr</code> and <code>FacLoad</code> options with default values of $1/\sqrt{2}$ define the scaling factors of (4.34) as $\zeta_u = \text{fu}/\sqrt{N}$ and $\zeta_\lambda = \text{fl}$ with $N$ the dimension of the solution.
<code>SphericalLin(Predic val(step); FacIncr fi; FacLoad fl; sim_par)</code>
This function uses the same parametric equation of <code>Spherical</code> but the linearized form (4.32) for its solution. The function parameters are the same as for <code>Spherical</code> . Here, it is important to correctly predict either the solution and/or the load parameter to avoid turning back on the solution path.

Table 3.19: Parametric equations for load control.

When computing families of solutions, simple continuation methods cannot pass turning or limit points, see the Section 4.3 *Non-Linear Solution Algorithms* for more details. Here load control or more properly arc-length continuation methods are used for stabilization. These methods are enabled with the option `LoadControl` together with the function `fun` and its default or required parameters as documented in Table 3.19. All these load control methods require the specification of the user parameter representing the load parameter. The parameter will be automatically updated and thus controlled by the load control algorithm requiring the residual derivative with respect to the load parameter specified with the option `DR_DSImPar`. If the derivative is left unspecified, we use the default function `Numerical(1.E-5)`. The user parameter specified with the `Solve Stationary` statement described on p. 80 is never the load parameter itself but the parameterization variable used by the load control algorithm. Its value is related to the length of the increments and load combined together.

## Command-statement `LinearSolver`



The incomplete LU factorization without additional fill-in. It is a good and robust preconditioner for M-matrices but may fail for other matrices. In this latter case, one may use a diagonal shift specified by the option `DiagShift` to stabilize the factorization, although in this way the approximation properties are getting worse. This preconditioner requires an additional matrix storage.

```
Eisenstat(DiagShift val; NoTrick)
```

This preconditioner is somewhat similar to the ILU one, it has the same `DiagShift` option but here the factorization is performed just on the diagonal and off-diagonal coefficients are left unchanged. Therefore the strictly upper and lower triangular matrices of the incomplete LU-factorization are the same ones of the unfactorized matrix and do not need to be stored. Without the `NoTrick` option, we use the Eisenstat's trick to save a matrix-vector product when applying the preconditioner. Here, the L-factor is applied as left preconditioner and the U-factor as right one so that a possible matrix symmetry is lost. Symmetry can be restored by assuming a positive definite matrix and a proper scaling of the L,U-factors, option not yet available. This preconditioner is generally inferior to ILU but just requires the storage of two additional vectors and can perform well for regular meshes.

```
Schwartz([<dom>[.<precond>(...)]+; BlockPrecond <precond>(...))
  SharedPrecond <precond>(...); SharedBlock;
  SharedSplit [<sdom>[.<precond>(...)]+)
```

This is the classical additive Schwartz domain decomposition preconditioner with minimal overlapping, a block preconditioner with disjointed blocks. The single subdomains are specified by the name `<dom>` of a user domain optionally followed by a dot and the block preconditioner to be applied, see Table 3.22. The notation `[ ]` represents an optional item and `[ ]+` one or more items. The definition's order is important since dofs can belong to just one block and dofs shared by several subdomains are associated to the first subdomain defining it. For dofs not included in any subdomain, no block preconditioner will be applied and the option `BlockPrecond` defines the default block preconditioner. With the option `SharedBlock`, the dofs shared by two or more blocks are collected in an additional block solved using the preconditioner specified by the option `SharedPrecond`. The same applies with the option `SharedSplit`, but the shared block is additionally split in several subblocks as defined by the list of user domains `sdom` optionally followed by a dot and the block preconditioner to be applied, if the block preconditioner is not given, the one specified by the option `SharedPrecond` is used.

```
Schur([<dom>[.<bprecond>(...)]+; BlockSolver <bprecond>(...);
  SchurSolver <solver>(...))
```

This is a Schur's complement preconditioner, a block preconditioner with shared subdomain dofs building up the Schur's complement block solved with a nested iterative solver. The subdomain selection as well as the block preconditioner selection is as for the Schwartz preconditioner, but here the shared subdomain dofs always define the Schur's complement block. The option `SchurSolver <solver>(...)` specifies the iterative solver for the Schur's complement block and available are the iterative solvers listed in Table 3.20.

```
Auxiliary(Projector[,1] [<dof-field>.<proj>]+; Matrix[,1] mat;
  Precond[,1] <bprecond>(...); Solver[,1] <solver>(...);
  PreSmooth n; PostSmooth n; ComplexSmoother)
```



An auxiliary preconditioner based on the solution of the back-stored system matrix `mat`. For each dof-field of the actual system block, one has to define the projection into the auxiliary space by the pair `<dof-field> . <proj>` where `<proj>` is a back-stored projection matrix assembled with the statement `Misc Matrix`. Before and after solving the auxiliary system with a nested solver `<solver>` as listed in Table 3.20 and with block preconditioner `<bprecond>`, `n`-step of the pre- and post- Gauss-Seidel symmetric smoother are performed. The option `ComplexSmoother` applies a  $2 \times 2$  block Gauss-Seidel smoother mimicking complex algebra. Two different and independent auxiliary systems can be solved in a multiplicative way, the second one is specified with the suffix `<...>1`.

None

No preconditioner is used.

Table 3.21: List of preconditioners. They all works with symmetric and non-symmetric matrices. Diagonal preconditioning is controlled by the option `DiagPrecond` and is applied as first.

This statement defines the linear solver used to solve the linear system of equations obtained by the discretization and eventually linearization of the governing partial differential equations defined within a block structure.

Without any option, the solver type `<solver>` is specified and a list of the most common used solvers is given in Table 3.20. Since the Kernel knows exactly the set of equations being solved, it can select the correct symmetric or unsymmetric matrix format, however, the user has the possibility to turn off this default selection with the options `Symmetric` or `NonSymmetric`. The option `CoeffCutOff` removes those matrix coefficients whose absolute value is less then the given value. Although the sparse matrix format is already optimized, some zero coefficients may still be present in the matrix which can be removed by this option with a zero cut-off value. This cut-off process is only applied when the matrix is constructed from scratch but not when a linear solution step is performed by reusing the previously computed sparse matrix format as for example the case when just user parameters are changed. Therefore it should be used with care, since coefficients which are zero in the first step, may not be zero anymore later on. The option `Reordering` defines the permutation of the equations. For a direct solver, it is used to reduce the fill-in and for an iterative solver to optimize an ILU preconditioner. The option `Compress` reduces the peak of memory usage occurring when assembling a matrix for the first time. The default zero value of `val` disables the memory optimization and increasing values of `val` in the useful range  $[0.2, 2]$  reduce the peak but increase the computational time. This option is only considered when using an iterative solver and is only opportune whenever the memory usage of the iterative solver is less than the peak occurring during the matrix construction.

The other options are just used by the iterative solver, if one has been selected. The option `DiagPrecond` symmetrically scales the system matrix to have absolut values of one in the diagonal. It is the most effective among all diagonal preconditioners, it does not use extra memory and therefore it is selected by default. The option `Preconditioner` selects the preconditioner used to speed-up the solver, they are listed in Table 3.21. The option `MaxIter` sets the maximal number of iterations and the solver returns an error if this number is exceeded. The option `InfoIter` prints



Direct
A direct solver based on Gauss elimination without pivot search but with a multiple minimum degree algorithm to reduce fill-in and therefore the amount of numerical operations.
CG(ConvergSolver fun(SOLVER); MaxIter n; InfoIter n)
The conjugate gradient solver. The options are the same ones of the LinearSolver statement, but have higher precedence and must be used whenever a nested iterative solver is used.
CGS(<CG-options>)
The conjugate gradient squared solver.
MINRES(<CG-options>)
The minimum residual solver.
BiCGStab(<CG-options>)
The stabilized biconjugate gradient solver.
BiCGStabL(<CG-options>; Ell dim; Delta val; ConvexHull)
The stabilized biconjugate gradient solver of order $l$ with accurate residual update. The order is specified by the option Ell (default 2) and whenever the residual norm is reduced by Delta, true residuals are newly computed. The option ConvexHull selects a more performant strategy for residual's reduction.
GMRES(<CG-options>; Krylov dim; Ortho <type>)
The general minimum residual solver. The option Krylov specifies the Krylov subspace dimension where residuals are minimized, default value is dim=10. The option Ortho specifies the used orthogonalization method among several variants of Gram-Schmidt.
Iterative(<CG-options>)
A short form to select a proper iterative solver. For symmetric matrices the CG solver is selected otherwise BiCGStab.
GaussSeidel(<CG-options>; nsym)
The Gauss-Seidel solver. The option nsym specifies the number of symmetric Gauss-Seidel steps before evaluating the true residuals.
ApplyPrecond(<CG-options>)
Apply once the (exact) preconditioner.
None
The linear system is just assembled but not solved.

Table 3.20: List of linear solvers. They all accept symmetric and non-symmetric matrices. The common options <CG-options> for all iterative solvers are redundant to the ones of the LinearSolver statement.

ILU(DiagShift val)
The same preconditioner ILU as in Table 3.21 is applied.
Eisenstat(DiagShift val)
The same preconditioner Eisenstat as in Table 3.21 is applied.
LU
The exact LU-factorization and the same solver as Direct in Table 3.20 is applied.
None
No block preconditioner is applied.

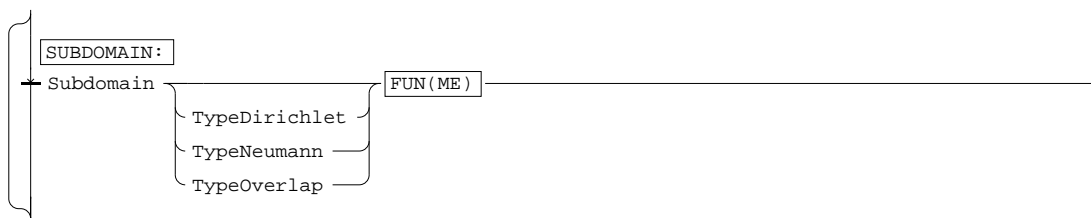
Table 3.22: List of block preconditioners.

Built-ins of ClassSOLVER	Description
NormR	The $L^2$ -norm of the new rhs-vector.
NormR0	The $L^2$ -norm of the initial rhs-vector.
nSolverIter	The number of performed solver iterations.

Table 3.23: Built-in symbols of ClassSOLVER available when defining convergence criteria for iterative linear solvers.

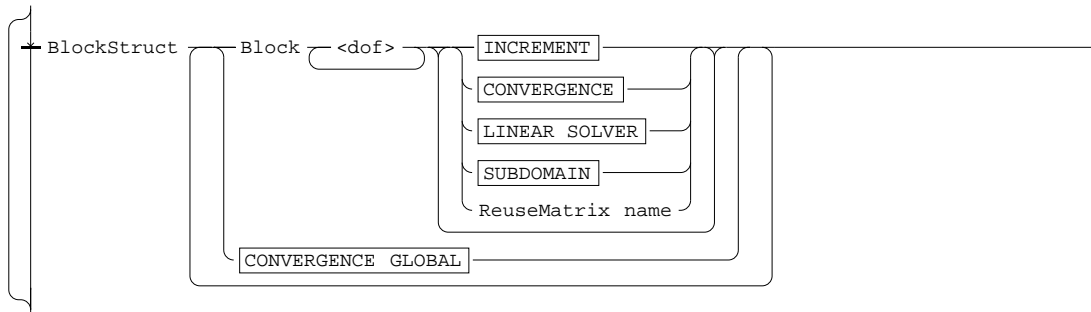
convergence infos each time after the number of given iterations has elapsed. The convergence of the iterative linear solvers is defined with the `ConvergSolver` option and the criterion is as a function of the input class `ClassSOLVER`, see Table 3.23. The convergence of iterative solvers is controlled by the  $L^2$ -norm of the rhs-vector that the iterative solver tries to reduce at each iteration. This value corresponds to the built-in `NormR` and the same value when the iteration solver is entered corresponds to `NormR0`. An absolute convergence criterion would then be formulated as e.g. `NormR<1.E-10` or a relative one as e.g. `NormR<1.E-2*NormR0`. When the equations to be solved are non-linear, a good practice is to perform a fixed and rather small number of iterations e.g. `nIterSolver>20` so that one always works with almost the latest update of residuals and tangent stiffness matrix. The rhs-vector norm `NormR` of the linear system is related to the residual block norms of the equations being solved, however, the linear solver uses another scaling. In particular the norm `NormR` is generally obtained after diagonal preconditioning and eventually an additional user selected right preconditioner. The options `UpLowMatrix` and `RectMatrix` determines the internal matrix storage format, the former using less memory and being slower than the latter.

## Command-statement `Subdomain`



This statement limits the solution of the discretized block equations to the domain where the given function does not evaluate to zero. This statement is mostly used to create matrices for numerical studies. With the default `TypeDirichlet` option, the dofs on the boundary in-between selected and unselected domain are removed, with the `TypeOverlap` option we keep the boundary dofs and with the `TypeNeumann` option we keep the boundary dofs but the contributions to these dofs just stem from the selected domain.

## Command-statement **BlockStruct**

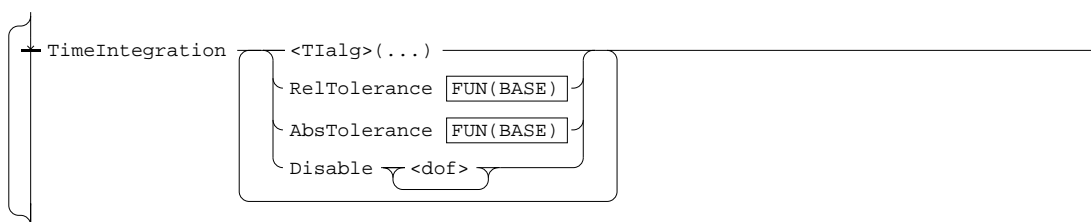


This statement specifies the list of block structures used to solve multiphysics problems. Here, the dof-fields are solved iteratively either using the Gauss-Seidel uncoupled algorithm, the Newton fully-coupled algorithm or a mixture of both algorithms. Dof-fields defined in a single block are solved with a fully coupled Newton's algorithm, whereas dof-fields defined in multiple blocks are solved with an uncoupled Gauss-Seidel algorithm, see also the Section [4.3 Non-Linear Solution Algorithms](#) for more details. Any list of blocks can be defined with some limitations on the dof-fields defined inside each block. However, if a block definition is accepted, it may be that not all dof-fields cross coupling terms in the computation of the derivatives are implemented in which case they are considered zero.

The **Block** option defines a new block structure containing the specified list of dof-fields `<dof>`. After the proper block definition, other relevant block data can be specified. The optional boxes **INCREMENT**, **CONVERGENCE**, **LINEAR SOLVER** and **SUBDOMAIN** are explained on p. [71](#), [69](#), [77](#), [73](#) and are used to set non-default values valid just for the current block. The option **ReuseMatrix** allows to reuse the constructed block system matrix that otherwise would be lost, each time another block structure is solved. At the end of the solution process for this block, the block system matrix is back-stored under the given name and later on at the beginning of the solution process for this same block, the back-stored matrix is reloaded, thus saving the process of constructing the same matrix again. This option is typically used to save LU-factorized matrices together with the option **ReuseFactoriz** of the **Increment** statement explained on p. [71](#), in order to avoid repeating the expensive numerical factorization. Back-stored matrices can be released with the statement **Misc MatrixRelease** explained on p. [86](#).

The optional box **CONVERG GLOBAL** is explained on p. [70](#) and is used to set non-default values valid just for this block structure.

## Command-statement **TimeIntegration**

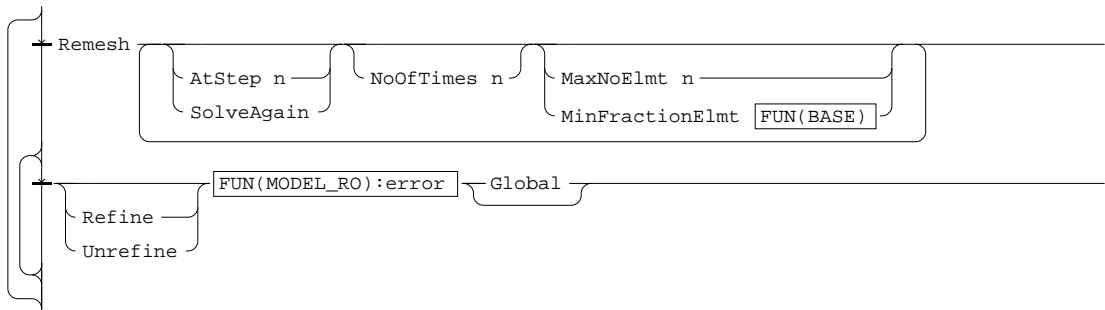


<TAlg>( ... )	Adaptive	Description
EulerBackward	No	Implicit Euler algorithm.
MidpointRule	Yes	Midpoint rule algorithm.
TrapezoidalRule(theta)	No	Generalized trapezoidal rule with parameter $\theta$ and $0.5 \leq \theta < 1$ .
BDF2	Yes	Backward differentiation formula of second order.
BDF3	No	Backward differentiation formula of third order.

Table 3.24: List of time integration algorithms.

This statement defines the time integration algorithm used when computing unstationary solutions. The list of available algorithms <TAlg>( ... ) with possible parameters is given in Table 3.24 and a detailed description is given in the Section 4.5 *Unstationary Solutions* and Table 4.2. If the time integration algorithm supports adaptive time step selection, the values of AbsTolerance and RelTolerance are used to control the accuracy of the integrated solution. They correspond to the values of  $\epsilon_{\text{abs}}$  and  $\epsilon_{\text{rel}}$  in the time step selection equation (4.45). The Disable option allows to disable the time integration algorithm for the given dof-fields. For a coupled problem where some dof-fields have a fast response, this is a better option than setting the mass matrix to zero or to a small value since no time integration is performed for the dof-fields.

### Command-statement Remesh

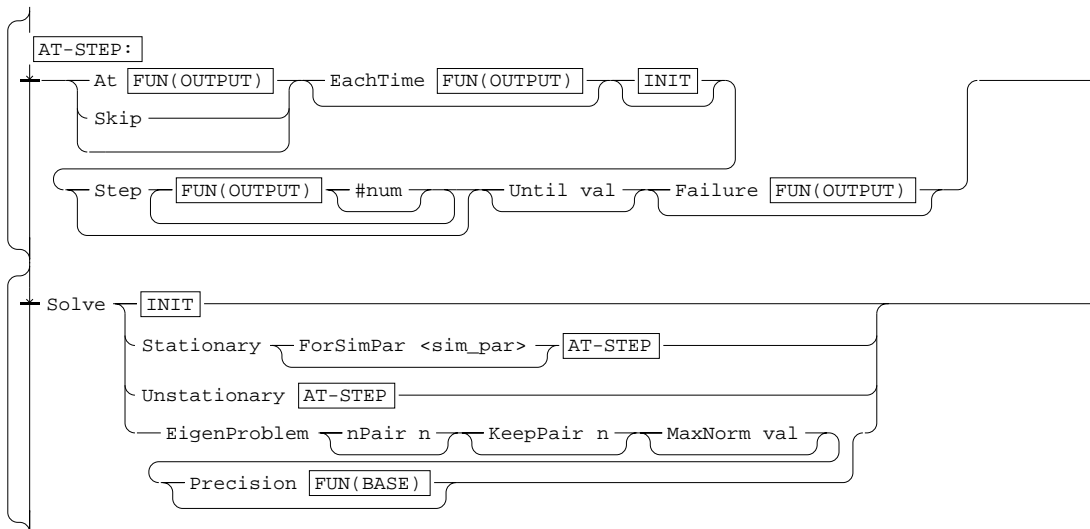


This statement refines and unrefines the element mesh according to a specified list of refinement criteria which mark the elements for refinement or unrefinement. When all refinement criteria have been evaluated, the elements are refined or unrefined according to the received mark whereby all defined dof-fields are best interpolated between the old and the new mesh. With the **AtStep** option, the action of remeshing is delayed and first performed at the end of every  $n$ -th computed stationary or unstationary solution step. To disable this automatic action do not specify any refinement criteria. The option **SolveAgain** is only used whenever the **AtStep** option is active and in this case the process of first solving and then remeshing is repeated indefinitely until the refinement criteria are fulfilled. With the option **NoOfTimes**, the remeshing operation is repeated  $n$ -times. With the **MaxNoElmt** option, an upper limit is set on the total number of generated elements and if the value **val** is exceeded, the refinement process stops thus preventing the waste of computational resources. With the op-

tion `MinFractionElmt` a lower limit on the number of newly generated elements is set and refinement stops whenever the fraction of newly generated elements divided by the number of elements is less than this value. This lower bound on the number of new elements is useful whenever the overall numerical error is already small and only new and strongly localized elements without overall error reduction will be generated. One then specifies a list of refinement criteria to mark the elements both for refinement or unrefinement. With the options `Refine` or `Unrefine`, elements are marked only for refinement or unrefinement. Without the `Global` option, elements are marked for refinement/unrefinement if the error estimator `error(MODEL_RO)` is greater/smaller than zero. With the `Global` option, those elements with the largest/smallest values of the error estimator are marked for refinement/unrefinement in order for the integral of the error estimator to be approximately zero. This type of adaptive refinement is particular useful when the accuracy of the fluxes through the contacts but not the accuracy of the dof-fields is relevant to the user. Since this global refinement strategy is less stringent than a local refinement strategy, less mesh elements are generally required to reach some accuracy goals for the contact fluxes.

When all refinement criteria have been evaluated, the elements are refined or unrefined according to the received mark. An element marked for refinement is always refined and if topological constraints with neighbor elements are violated, the offending neighbor elements will be refined as well. If the element is marked for unrefinement, the element will be unrefined only if all elements in the same family are marked for unrefinement and if topological constraints with neighbor elements are not violated. If an element is marked both for refinement and unrefinement, the element will be refined, see also the Section 4.1 *Finite Element Mesh* for more details on topological mesh constraints. After execution of this statement, the control variable `REMESH` is set to zero if the mesh has been modified, otherwise to a non-zero value. If the value is one, all refinement criteria were fulfilled and by other non-zero values the refinement has been stopped by the user, e.g. with the option `MaxNoElmt`.

## Command-statement `Solve`



This statement specifies solution algorithms. If during execution of this statement numerical errors occur or the function `failure` is called by the user, see Table 3.3, execution of this statement is stopped and one proceeds with next statement. If execution is interrupted, the global control variable `SOLVE` will be set to one otherwise to zero. Numerical errors may be trapped and execution may proceed if one defines what to do by numerical failures.

The `INIT` option initializes all dof-field vectors. Per default the vectors are initialized to zero, but other values may be defined as described on p. 68 with the `Init` statement. Initial values defined here are only valid for this initialization and temporarily replace values set with the `Init` statement.

The `Stationary` option computes stationary solutions at different values of the actual simulation parameter selected with the option `ForSimPar`. The parameter can be the pseudo time `time` (default value) or a component of any user parameter, see also the Sections 3.2 *User Parameters* and 4.3 *Non-Linear Solution Algorithms*. The values of the simulation parameter used in computing the solutions are selected with the box `[AT-STEP]`, where the actual value, the last selected increment and the number of performed solutions can be accessed by the global variables `value`, `step` and `n_step` defined in the Table 3.3. For the first solution, the user can set the initial value with the `At` option, otherwise the simulation parameter value is left unchanged. With the `Skip` option, the computation of the first solution is skipped meaning that a valid solution is already available and one can proceed with the next solution. The option `EachTime` is used to call this function each time before starting a solution, however, the returned value is not used by the system and the call is typically used to print messages. The option `INIT` is used to initialize dof-fields which are not solved by this statement but are dependent on the simulation parameter and need to be initialized before starting a solution. If several stationary solutions should be computed, the user specifies with the `Step` option the increment of the simulation parameter, optionally followed by the repetition symbol `#` and the number `num` of repetitions. If the step function evaluates to null the step iteration is stopped. With the `Until` option, solutions are computed only up to this final value of the simulation parameter and if the repetition value for the last step is not specified, the last incremental step is indefinitely repeated until reaching the final value. With the `Failure` option, if numerical errors during the solution occur or the user calls the function `failure`, the actual step is discarded and repeated with this new step value, otherwise the solution process stops and the control variable `SOLVE` is set to one. If the function `failure` is called again within the failure call, execution stops and the variable `SOLVE` is set to one. Failure conditions are typically defined together with convergence criteria and the statements `Convergence` and `ConvergGlobal` explained on p. 69 and p. 70. If several solutions steps are computed, it is possible with the `Dump` statement explained on p. 83, to automatically append data to a data file at regular solution step intervals.

The `Unstationary` option computes unstationary solutions. Here, solutions are computed at different values of the simulation time, whereas all other user parameters are left unchanged, see also the Section 3.2 *User Parameters*. The time integration algorithm used to compute evolutionary solutions is selected with the `TimeIntegration` statement explained on p. 78; a description of these algorithms is given in the Section 4.5 *Unstationary Solutions*. The selection of the time step is done similarly as



for the computation of stationary solutions, however, the option `ForSimPar` is not available here and at the initial time no solution has to be computed. Instead the user should have defined a proper initial solution and solutions are computed starting from the first time step. This behavior corresponds to the `Skip` option of the stationary case. When defining the step, you can use the global variable `automatic_step` to automatically select the next increment which is computed by monitoring and comparing the local truncation error with the tolerance values `RelTolerance` and `AbsTolerance` defined by the `TimeIntegration` statement. If the integration algorithm does not support an automatic step selection, the last step value will be used. Since none of the implemented automatic algorithms are self-starting, you always have to provide an initial step not defined by `automatic_step`. In general, the integration algorithms require an initialization phase where lower order methods are used, so that if one restarts the integration some accuracy is lost, except if one restarts with very short time steps. However, when ending the execution of the `Solve Unstationary` command, the data required by the integration algorithms is kept stored in memory so that the integration can be continued with the next `Solve Unstationary` statement but only if the `At` option is not used and if other `Solve` statements have not been issued. If the time integration is continued, the value `automatic_step` can be used as first step and represents the last computed value.

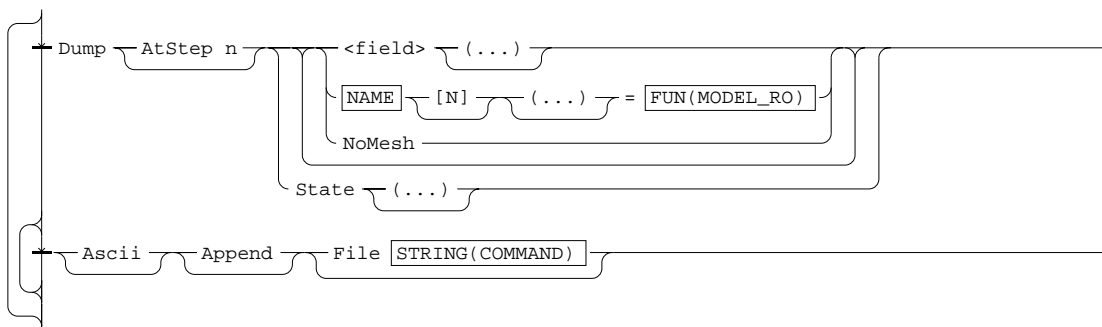
The `EigenProblem` option computes some eigen-pairs i.e. eigenvalues and eigenvectors of the Helmholtz problem associated with the defined block structure. In general, but this is not always true, the computed eigen-pairs are the eigen-pairs of the system matrix defined by the first block structure at the actual solution time i.e. the eigen-pairs of the first system matrix to be inverted when using the `Stationary` statement. For details on the Helmholtz problem being solved, we refer to Chapter 5 *Physical Models*. Some restrictions apply when computing eigen-pair solutions. A direct solver must be selected, see p. 73 and the system matrix defined by the block structure must be symmetric. Further, inhomogeneous Dirichlet, Neumann and floating boundary conditions are not considered; if they are defined they will be converted to homogeneous boundary conditions and this without generating error messages. The `nPair` option specifies the number of eigen-pairs to be computed, where the pairs are ordered with increasing eigenvalues. Without this option, only one eigen-pair is computed. Per default, the first computed eigen-vector will replace the dof-fields defined in the block structure, but with the `KeepPair` option the `n`-th eigen-vector will be used for the replacement. The `Precision` option determines the precision of the computed eigenvalues. The eigenvectors are normed to one with respect to the norm induced by the positive definite mass matrix. With the `MaxNorm` option, the norm can be changed so that the maximal value of the eigenvector components is the given value. If the `Write` or the `Dump` statement explained on p. 83 have been used to activate delayed writing of data files, this action is applied for each computed eigen-pair.



Parameter	Description
Unit <small>UNIT</small>	The unit for the output field.
Title <small>FORMAT</small>	A title for additional information.
Discontinuous	This is the default numerical format with a local element field approximation resulting in a possibly discontinuous field between elements.
Continuous	The local element field approximations are averaged between elements resulting in an overall continuous field. This option reduces the size of the data file but should be used with care since many numerical fields are not continuous.
Displacement	The field may be used as a displacement field for displaying a displaced mesh.

Table 3.25: List of parameters for output fields.

### Command-statement Dump



With this statement graphics and state data files are written by the Kernel program. Graphics data files are written for visualization purposes to pass mesh and field information from the Kernel program to the Front End program, whereas state data files are written to save the intermediate state of a simulation. State files can then be loaded by the Kernel program to continue a computation from the point of writing, e.g. a previously generated mesh file can be reloaded into the Kernel program to save the effort of refining a mesh from scratch.

With the `AtStep` option, the action of writing the file is delayed and first performed at the end of every  $n$ -th computed stationary or unstationary solution step. This is also true when computing eigen-pairs, but here the parameter `AtStep` has no meaning and the writing of a data file is applied to every computed eigen-pair. See also the `Solve` statement explained on p. 80. To disable the automatic action of dumping data do not specify anything to dump.

When dumping a data file, the default operation is to write a graphics data file used later by the Front End program to visualize numerical fields. Here, the user specifies with `<field>` any built-in symbol of `ClassMODEL`, see Table 3.10, optionally followed by a list of parameters `(...)` to change the default format, see Table 3.25. With the option `[NAME]`, the user can specify a new field carrying that name, of dimension  $N$  and defined by a function of the input class `ClassMODELRO`, see Table 3.10. As for the definition of arrays, instead of the dimension one can also specify a list of components name, and the same parameters of Table 3.25 apply. Per default the mesh data is added to the file, however, if several slots are written and the mesh has not

changed, one can save disk space by excluding to write the same mesh data again with the `NoMesh` option. We notice that material parameters which are independent from computed fields can be quickly displayed by the Front End program without the need to write them into data files.

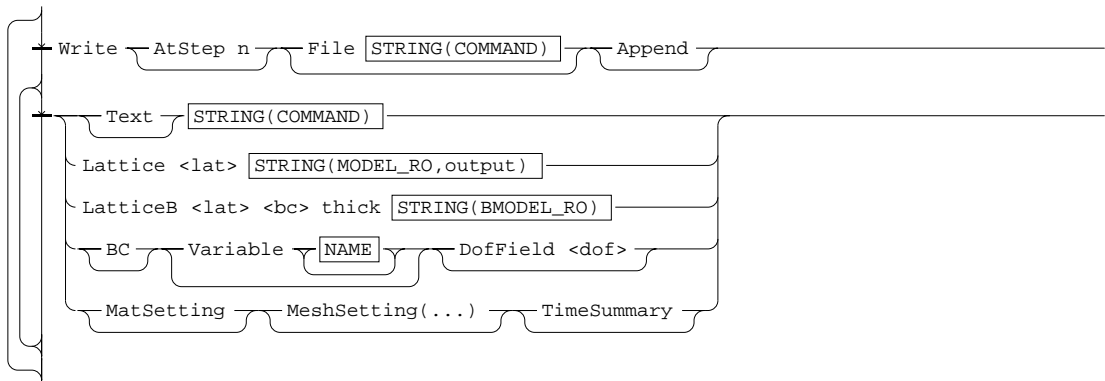
The `State` option allows the user to write a state data file, that later on is reloaded to continue the numerical simulation from the point of writing. To save disk space, you may specify the parameters `NoDofField`, `NoUserField` and `NoMesh`. Per default, the file does not store user global variables, however, they may be added by declaring the parameter `Variable` followed by a list of global variables to be stored.

The `Append` option is used to append the data to the file instead of creating a new empty one. This option is very useful when visualizing a field that changes its values during a computation, as for example, when computing unstationary solutions. In this case, the Front End program allows a fast repeated visualization of the field at different times thus creating an animation effect. Note, however, that a state data file cannot be appended to a graphics data file and viceversa. In the case the `Append` option is used together with the `AtStep` option to enable delayed writing of data files, the append operation only refers to the first write operation, for all subsequent ones it is automatically implied.

The default data format is a compact machine-dependent binary format. To ensure portability among different computer platforms, data files should be written in the ASCII format using the option `Ascii`, which however makes them much larger. In this form, they can also be viewed with a normal text editor. For state data files is recommended to use the binary format which represents a true copy of the memory content. With the ASCII format the input-output conversion of numerical values between the binary and decimal notation may lead to small differences in the data.

Each graphics data file is given the default name `Data` and each state data file has the default name `State`; any other name can be specified with the `File` option.

## Command-statement Write



This statement prints text to the output stream. With the `AtStep` option, the action of printing is delayed and first performed at the end of every `n`-th computed stationary or unstationary solution step. This is also true when computing eigen-pairs, but here the parameter `AtStep` has no meaning and the writing of a data file is applied to every

computed eigen-pair. See also the `Solve` statement explained on p. 80. To disable the automatic action of printing do not specify anything to write.

With the `File` option, the output is redirected to the specified file until the next `File` option is issued. With the `Append` option the data will be appended to the file instead of starting with a new empty file. If the `AtStep` option is used, the append operation only refers to the first write operation, for all subsequent ones it is automatically implied.

With or without the option `Text`, the text `[STRING]` constructed with numerical values of the input class `ClassOUTPUT`, see Table 3.13 and Section 3.3 *Literals and Strings*, is printed to the output. With the option `Lattice` the same applies, but the numerical values are of the input class `ClassMODELRO`, see Table 3.10 and with the addition of the built-in output of class `ClassOUTPUT`, see Table 3.13. Here, the numerical values are evaluated and printed at every point of the lattice `<lat>` defined with the `Lattice` statement explained on p. 60. The evaluation order strictly follows the lattice point definition and working with global variables is thread save and thus permitted. For example, to print the  $x$  and  $y$  coordinates of the lattice `lat` use the statement `Write Lattice lat "x=%g y=%g\n" x y`. The option `LatticeB` is similar to `Lattice` but the numerical values are of the input class `ClassBMODELRO`, see Table 3.11 and evaluation is just performed for lattice points lying within the volume formed by the boundary's surface of `<bc>` times the transverse thickness `thick`. The option `BC` prints the boundary condition fluxes for the actual solution in a default format. The option `Variable` prints the values of the listed global variables and all ones if no name is given. The option `DofField` prints the numerical dof-field `<dof>`. The option `MatSetting` prints the values of constant material parameters. The option `MeshSetting` prints the mesh in a plain *SESES* format so that it can be imported again and used by the Front End's mesh builder. With the argument `WithDisp`, the actual mechanical displacement is added to the ME coordinates. The option `TimeSummary` prints the computational time for various tasks and reset all timers so that by the next call, the new incremental times are displayed.

## Command-statement Load

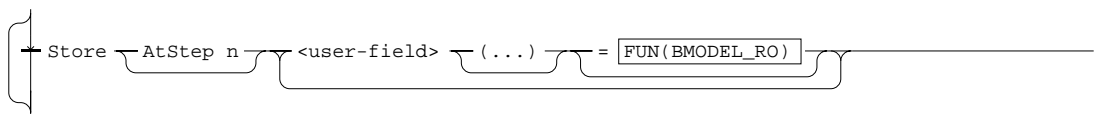


This statement loads a previously written state data file into the Kernel program. If the `Append` option was used at least  $n$  times when writing the file, the `Slot` parameter allows the selection of the  $n$ -th written slot state, instead of the first slot selected by default. Note that if the slot does not define the mesh data i.e. it has been written with the option `NoMesh` and the mesh in the Kernel program is incompatible with that of the file, the dof-field data cannot be loaded.

Parameter	Description
DofField <dof>+	A list of dof-fields. The element field is defined if at least one dof-field is locally defined.
ForMaterial <mat>+	A list of materials. The element field is additionally defined just on the specified materials.
IpFrom [FIELD, SET0, SET1, SET2]	The set of integration points where the element field is defined within an element. The FIELD option takes the set generally used by one of the dof-field above and should be used if just one dof-field is specified. The other options select built-in sets listed with an increasing integration accuracy. The set SET0 defines a single point element field which always has read-write permissions, but its declaration must be done in the initial section, see p. 49.
Continuous fun(ME)	The element field is smoothed over the element boundaries of the domain fun(ME). In other words, the element field generally computed and stored at some element integration points is modified so that extrapolation to the element boundary is continuous among neighbor elements.
FreeOnRef	Whenever the mesh is refined or unrefined, the element or boundary field is released instead of being interpolated on the new mesh.
RestoreOnFailure	Whenever a solution is discarded because of a call to the function failure of Table 3.3, the original value of the field before the discarded solution step is restored.
KeepForUndefDof	Element fields are associated with dof-fields and whenever these dof-fields are undefined, the element fields are released too, except if this option is used.

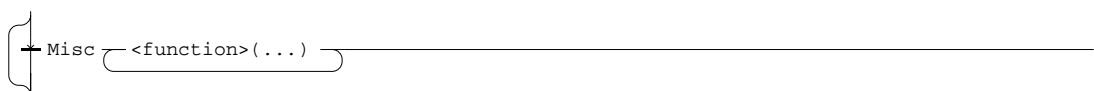
Table 3.26: List of available options for storage of user fields.

## Command-statement Store



This statement defines the element or boundary field `user-field` declared with the statements `ElmtFieldDef` and `BoundFieldDef` explained on p. 49. The field is defined by a function of the input class `ClassBMODEL_RO`, see Tables 3.11 and if not previously defined, memory will be allocated. If no function is given, the memory is released and subsequently the field will have a zero value. Optionally a list of parameters `(...)` to modify the default storage can be given, see Table 3.26. With the `AtStep` option, the action of defining the element or boundary fields is delayed and first performed at the end of every `n`-th computed stationary or unstationary solution step. To disable this automatic action, do not specify anythings to store.

## Command-statement Misc



This statement collects together all those singularly programmed tasks which other-

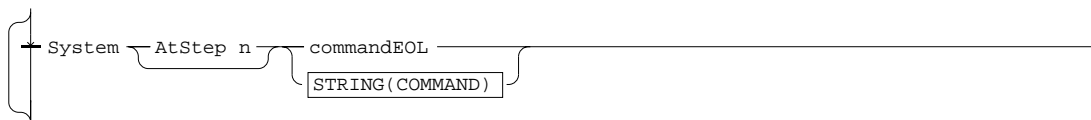
wise cannot be performed with the other statements. The tasks are executed by calling the built-in function `function` and the available choices together with default or required parameters are documented in Table 3.27.

### Command-statement **Extract**



This statement is only processed when a container file is used for input and it extracts the file `LITERAL` from the input file into the actual working directory, see also the Section 1.6 *Embedding files in a single container file*. This option is used together with the statement `System` in order to create input files to be used within the executed commands. In this way, it is possible to run *SESES* and general system commands by having everything defined in a single file i.e. the input container file.

### Command-statement **System**



This statement allows the execution of system commands. With the `AtStep` option, the action of execution of the command is delayed and first performed at the end of every  $n$ -th computed stationary or unstationary solution step. To disable this automatic action, do not specify anything to execute. The command to be executed may be specified in two forms. In the first form `commandEOL`, the command is taken to be the full literal text up to the first end-of-line character, i.e. all type of text preprocessing is disabled here, however, by escaping the end-of-line character with a backslash, commands can be continued on the next input line. The second form uses a string `STRING` constructed with numerical values of the input class `ClassCOMMAND`, see Table 3.13 and Section 3.3 *Literals and Strings*.

Input data to the executed commands is better defined through input files, since in general single line commands do not permit a definition of the data. If all simulation files should be packed in a single input container file, see the Section 1.6 *Embedding files in a single container file*, these input files can be extracted before running the command with the `Extract` statement.

<code>DR_DSimPar(nIter niter; Err err; eps)</code>
This function verifies the correct definition of the residual derivative with respect to a user parameter, where the derivative is defined with the statement <code>Increment DR_DSimPar</code> explained on p. 71 together with the functions of Table 3.18. The derivative is checked numerically by running a Newton's algorithm; if the convergence behavior is quadratic than to a high degree of confidence it is also correct or at least good enough to run algorithms requiring the derivative. In order to start the test, one has first to compute a solution by fixing the value of the user parameter where the derivative is checked. Assuming zero residuals, this function is called and the value <code>eps</code> is added to the user parameter. The residuals are not zero anymore and Newton's algorithm for the single scalar equation represented by the sum of all residual components is started to obtain again zero residuals. At each iteration of a maximum of <code>niter</code> , the convergence behavior is shown on the output. If at the iteration's end the residual error exceeds <code>err</code> and error is reported. The default values are <code>nIter=15</code> and <code>Err=0</code> . In many cases, the dependency from the user parameter can be chosen to be linear, so convergence requires one single step.
<code>Matrix(&lt;dof-field&gt;; &lt;matrix&gt;; name)</code>
This function allows to back-store matrices not available as block system matrices with the <code>BlockStruct</code> statement. These matrices are generally non-square matrices representing projections or prolongation matrices used when constructing preconditioners. The matrix selected from the list <code>&lt;matrix&gt;</code> is back-stored under the name <code>name</code> . The range of the matrix is represented by the <code>dofs</code> of the actual block and the domain of the matrix are the <code>dofs</code> of the <code>dof-field &lt;dof-field&gt;</code> modified as determined by the type of matrix <code>&lt;matrix&gt;</code> .
<code>MatrixRelease(Matrix mat)</code>
Release the back-stored matrix <code>mat</code> or all ones if no name is given, see also the <code>BlockStruct</code> statement on p. 78.
<code>RigidBody([block &lt;surface&gt;]+); LinearSearch; Clear; Smooth; Extrusion val)</code>
This function declares a rigid-body used by the built-in routine <code>RigidIntersection</code> for mechanical contact detection, see p. 141. The rigid-body is defined by the list <code>[block &lt;surface&gt;]+</code> with <code>block</code> a block of hexahedral MEs and <code>&lt;surface&gt;</code> the hexahedral contact's surface. In the surface transverse direction, the ME block must consist of a single ME. With the <code>LinearSearch</code> option, a slow linear search not requiring any additional memory is done, otherwise an octree based search. With the <code>Clear</code> option, any defined rigid-body is cleared. By default, the lowest order interpolation of the ME nodal coordinates is performed so that normal and tangent vectors as returned by <code>RigidIntersection</code> are generally discontinuous and curvatures are unevaluated. With the <code>Smooth</code> option, cubic hermitic interpolation is performed returning continuous values of normal and tangent vectors. If smoothing is defined, the points of the unsmoothed surface are shifted along the transversal ME direction according to the selected smoothing function. The <code>Extrusion</code> value enlarges the ME with respect to its center of mass when performing the ME search. This is just needed when smoothing is enabled, since the initial ME search is always performed with respect to the native ME coordinates and smoothed points may lie outside these MEs.

Table 3.27: Miscellaneous task functions.

## Chapter 4

# Numerical Models

This chapter reviews the fundamentals of multiphysics modeling, finite element theories and related numerical algorithms available within *SESES*. Only the basics in abbreviated form are given here, for a more in-depth understanding the interested reader should consult among others [1, 2, 3, 4, 5, 7, 11, 14].

Multiphysics modeling is generally associated with transport and electrodynamics phenomena at the macro and micro length scales which are coupled together. In particular, a continuum hypothesis at the molecular scale is assumed ruling out quantum fluctuations associated with the nano scale and below. From labor experiments on charged and moving particles, we obtain the governing equations of electrodynamics which for continuous media, they are given by the macroscopic Maxwell's equation

$$\begin{aligned} \nabla \cdot \mathbf{D} &= \rho, & \nabla \times \mathbf{H} - \partial_t \mathbf{D} &= \mathbf{J}, \\ \nabla \times \mathbf{E} + \partial_t \mathbf{B} &= 0, & \nabla \cdot \mathbf{B} &= 0, \end{aligned} \quad (4.1)$$

with  $\rho$  the charge density,  $\mathbf{J}$  the electric current density,  $\mathbf{E}$  the electric field,  $\mathbf{B}$  the induction field,  $\mathbf{D}$  the electric displacement and  $\mathbf{H}$  the magnetic field all being functions of  $(\mathbf{x}, t)$  with  $t$  the time and  $\mathbf{x} \in \mathbb{R}^3$  the space variable. The fields  $\mathbf{D} = \mathbf{D}(\mathbf{E}, \mathbf{H})$  and  $\mathbf{B} = \mathbf{B}(\mathbf{E}, \mathbf{H})$  are defined by material laws, but the dependency is generally linear and given by  $\mathbf{D} = \epsilon \mathbf{E}$  and  $\mathbf{B} = \mu \mathbf{H}$  with  $\epsilon$  the electric permittivity or dielectricity and  $\mu$  the magnetic permeability.

Another typical governing equation of multiphysics is the one associated with the conservation of a measurable quantity  $\alpha(\mathbf{x}, t)$  transported by an ensemble of moving particles inside a domain  $\Omega \subset \mathbb{R}^3$  with  $\mathbf{x} \in \Omega$ . For its derivation, at  $t = 0$  one first assumes the existence of an initial domain  $\Omega_0 \subset \mathbb{R}^3$  with all particles at rest, secondly together with the continuum hypothesis, at  $t > 0$  one assumes the particle motion to be given by a one-to-one smooth map  $\mathbf{x} = \varphi(\mathbf{X}, t)$  for any particle originally located at  $\mathbf{X} \in \Omega_0$ . By considering any control volume  $\omega \subset \Omega$  moving with the particles  $\omega = \varphi(\omega_0)$  and  $\omega_0 \subset \Omega_0$  a control volume at rest, an axiomatic balance law for  $\alpha(\mathbf{x}, t)$  can be stated as  $(d/dt) \int_{\omega} \alpha = \int_{\partial\omega} \beta + \int_{\omega} \gamma$  with  $\gamma(\mathbf{x}, t)$  a local rate of change associated with  $\alpha(\mathbf{x}, t)$  and  $\beta(\mathbf{n}, \mathbf{x}, t)$  a quantity representing intermolecular transport phenomena across the boundary  $\partial\omega$  and therefore also depending on the outward normal  $\mathbf{n}$  to the boundary  $\partial\omega$ . Physical evidence will tell us about the three quantities  $(\alpha, \beta, \gamma)$ , but before giving examples let's develop the matter further. As first, by a dimensional analysis



Cauchy theorem tells us that for  $\beta$ , we necessarily must have a linear dependency of the form  $\beta(\mathbf{n}, \mathbf{x}, t) = \beta(\mathbf{x}, t) \cdot \mathbf{n}(\mathbf{x})$  with  $\beta$  a vector valued flux. Secondly, by evaluating the time derivative  $(d/dt) \int_{\omega} \alpha$  and applying Gauss theorem on the boundary integral  $\int_{\partial\omega} \beta \cdot \mathbf{n}$ , we obtain the integral form  $\int_{\omega} [\partial_t \alpha + \nabla \cdot (\alpha \mathbf{v})] = \int_{\omega} [\nabla \cdot \beta + \gamma]$  with  $\mathbf{v} = \partial_t \mathbf{x}$  the flow velocity. The relation  $(d/dt) \int_{\omega} \alpha = \int_{\omega} [\partial_t \alpha + \nabla \alpha \cdot \mathbf{v} + (\nabla \cdot \mathbf{v}) \alpha]$  is known as transport theorem, the first two terms on the right-hand side stem from the time-derivative  $(d/dt) \alpha(\mathbf{x}, t) = (d/dt) \alpha(\varphi(\mathbf{X}, t), t)$  under the integral and the third one by considering the time-derivative on the moving domain  $\omega = \varphi(\omega_0)$ . Since the control volume  $\omega$  is generic, for  $\mathbf{x} \in \Omega$  we obtain the differential form of a generic scalar transport law as

$$\partial_t \alpha + \nabla \cdot (\alpha \mathbf{v}) = \nabla \cdot \beta + \gamma. \quad (4.2)$$

The first example of a conservation law is given by the mass density  $\alpha = \rho$ . In classical physics the mass is conserved  $\gamma = 0$  and since here the mass is exactly moving with the flow, we also have  $\beta = 0$  resulting in the continuity mass equation

$$\partial_t \rho + \nabla \cdot (\rho \mathbf{v}) = 0. \quad (4.3)$$

A second similar example is given by a tracer  $\alpha = c$  exactly following the flow and having a source/sink rate of  $\gamma = \Pi$ , again we have  $\beta = 0$  resulting in  $\partial_t c + \nabla \cdot (c \mathbf{v}) = \Pi$ . A third example is given by considering single components of the mechanical momentum  $\alpha = \rho v_i$  with  $i = x, y, z$ . In this case, we have  $\beta = \mathbf{s}_{i\bullet}$  and  $\gamma = \mathbf{f}_i$  with  $\mathbf{s}$  the stress tensor and  $\mathbf{f}$  the body force resulting in

$$\partial_t (\rho \mathbf{v}) + \nabla \cdot (\rho \mathbf{v} \otimes \mathbf{v}) = \nabla \cdot \mathbf{s} + \mathbf{f}. \quad (4.4)$$

With the help of the continuity mass equation, this conservative form can also be rewritten in the non-conservative form as  $\rho \partial_t \mathbf{v} + \rho (\nabla \mathbf{v}) \mathbf{v} = \nabla \cdot \mathbf{s} + \mathbf{f}$ . For a fluid, the specification of the stress  $\mathbf{s} = \mathbf{s}(\nabla \mathbf{v})$  as a function of the velocity gradient  $\nabla \mathbf{v}$  will result in the Navier-Stokes equations, whereas for an elastic solid, the specification of the stress  $\mathbf{s} = \mathbf{s}(\mathbf{F})$  as a function of the deformation gradient  $\mathbf{F} = \partial \mathbf{x} / \partial \mathbf{X}$  results in the equations of elasticity. Similarly to the momentum, we can also consider the angular momentum  $\alpha = (\rho \mathbf{v} \times \mathbf{x})_i$  resulting in the symmetry of the stress tensor  $\mathbf{s}$ . Another important example is given by  $\alpha$  being some thermodynamical extensive state variables like the internal energy or the enthalpy. Depending on the system under investigation, thermodynamics will tell us the particular form of  $(\alpha, \gamma)$  with  $\beta$  being the thermal conductive flux.

We have given some examples of governing equations typically arising when modeling multiphysics systems. The Maxwell's equations (4.1) and the generic transport law (4.2) build a system of first order PDEs, however, with the introduction of electromagnetic potentials for the Maxwell's equations and the closure of the transport equations by defining material laws, we generally obtain PDEs with first and second order derivatives. By the nature of the multiphysics system under investigation many of these equations may be coupled together and Table 4.1 gives some typical examples of conservation laws, material laws and coupling terms. Numerically, it is not possible to solve these equations for the Euclidian space  $\Omega = \mathbb{R}^3$  so that one has to consider bounded domains  $\Omega \subset \mathbb{R}^3$  and in the case of some axis symmetry of 2D domains  $\Omega \subset \mathbb{R}^2$ , 1D domains are not considered in SESES. The interaction with the external world  $\Omega^c = \mathbb{R}^{3,2} \setminus \Omega$  needs to be taken into account by the defining boundary

	Type of Law	Equation	Description
$\mathcal{E}_1$	Conservation law	$\nabla \cdot \mathbf{J} = 0$	Charge conservation for the electric current density $\mathbf{J}$ .
$\mathcal{M}_1$	Ohm's law	$\mathbf{J} = \sigma \mathbf{E} = -\sigma \nabla \psi$	Infinitesimal Ohm's law, with $\sigma$ the conductivity, $\psi$ the electric potential, and $\mathbf{E}$ the electric field.
$\mathcal{E}_2$	Conservation law	$\nabla \cdot \mathbf{F} = q$	Thermal energy conservation for the heat flux $\mathbf{F}$ including the heat generation rate $q$ .
$\mathcal{M}_2$	Fick's 1st law	$\mathbf{F} = -\kappa \nabla T$	The heat flux $\mathbf{F}$ is proportional to the temperature gradient. $T$ denotes the temperature distribution and $\kappa$ the thermal conductivity.
$\mathcal{E}_3$	Conservation law	$\nabla \cdot \mathbf{s} = \mathbf{f}$	Linearized momentum conservation for the mechanical stress tensor $\mathbf{s}$ in the presence of a body force $\mathbf{f}$ .
$\mathcal{M}_3$	Hook's law	$\mathbf{s} = \mathbf{C} \cdot (\boldsymbol{\epsilon} - \boldsymbol{\epsilon}_{\text{ini}})$ $\boldsymbol{\epsilon} = \frac{1}{2}(\nabla \mathbf{u} + (\nabla \mathbf{u})^T)$	The stress tensor $\mathbf{s}$ is given in terms of the elasticity tensor $\mathbf{C}$ , the strain tensor $\boldsymbol{\epsilon}$ and initial strain $\boldsymbol{\epsilon}_{\text{ini}}$ with $\mathbf{u}$ the deformation.
$\mathcal{C}_1$	Joule's heat	$q = \sigma  \nabla \psi ^2$	The heat source term depends on the electric field, electric( $\mathcal{E}_1$ ) $\rightarrow$ thermal( $\mathcal{E}_2$ ) coupling.
$\mathcal{C}_2$	Thermally induced strains	$\boldsymbol{\epsilon}^0 = \alpha(T)$	The initial strain tensor is temperature dependent with $\alpha$ the thermal expansion tensor, thermal( $\mathcal{E}_2$ ) $\rightarrow$ mechanic( $\mathcal{E}_3$ ) coupling.
$\mathcal{C}_3$	Temperature coefficients	$\sigma = \sigma(T)$	The electric conductivity is temperature dependent, thermal( $\mathcal{E}_2$ ) $\rightarrow$ electric( $\mathcal{E}_1$ ) coupling.
$\mathcal{C}_4$	Piezoresistivity	$\sigma = \sigma(\mathbf{s})$	The mechanical stress tensor $\mathbf{s}$ alters the electric conductivity, mechanic( $\mathcal{E}_3$ ) $\rightarrow$ electric( $\mathcal{E}_1$ ) coupling.
$\mathcal{C}_5$	Change of the domain	$\mathbf{u} = \mathbf{u}(T)$	In the presence of large deformations a change of the modeling domain occurs, mechanic( $\mathcal{E}_3$ ) $\rightarrow$ electric( $\mathcal{E}_1$ ), thermal( $\mathcal{E}_2$ ) coupling.

Table 4.1: Some typical conservation laws  $\mathcal{E}_1 - \mathcal{E}_3$ , material laws  $\mathcal{M}_1 - \mathcal{M}_3$  and coupling mechanisms  $\mathcal{C}_1 - \mathcal{C}_5$ — available in *SESES*.

conditions (BCs) to hold on the domain boundary  $\partial\Omega = \overline{\Omega} \cap \Omega^c$  where the governing equations are to be solved. Since in general we are dealing with 2nd order PDEs, one has either Dirichlet BCs prescribing the value of primary dof-fields or Neumann BCs prescribing some related secondary flux-field. Clearly is not always possible to model the complexity of the exterior  $\Omega^c$  just by specifying BCs and here an enlargement of the domain should be considered.

When computing numerical solutions, the first step is to discretize in space and time the governing equations. For the spatial discretization, the most prominent techniques are finite elements and finite volumes methods where the computational domain is first decomposed into an admissible mesh of non-overlapping finite elements or finite volumes of some simple geometrical shape. At the present time, *SESES* exclusively uses finite element methods, but many finite element types are generally available and so a proper choice must be based on properties as numerical accuracy and computational efficiency. Once these technicalities have been addressed, the implementation of the finite element method does not in general present major problems. However, the important issue of constructing a suitable computational mesh is left to the user. In *SESES* we can have a two step approach, where the user first define a coarse initial mesh for the purpose of defining material domains and boundary conditions only and where the mesh is adaptively refined afterwards in order to improve the accuracy of the solution based on some a posteriori error estimator.

After the discretization of the governing equations with the inclusion of the required coupling terms, a system of non-linear equations must be solved. The literature offers many approaches to this problem, including the non-linear block Gauss-Seidel and the fully coupled Newton's algorithms. In *SESES* a generalized block solution algorithm is available which includes both algorithms as special cases. Of course, the complexity of computing solutions of multiphysics coupled problems is largely increased with respect to solving single governing equations and the mathematical proofs on the existence or uniqueness of solutions are generally missing.

## 4.1 Finite Element Mesh

In this section, we present the finite element meshes available in *SESES* and discuss some issues related to their utilization. The finite element mesh is the basic ingredient of any finite element method and its purpose is to partition the simulation domain  $\Omega$  into simple polyhedral shapes  $\Omega^e$  called *elements* where low order field approximations are defined.

As described in the Chapter 3 *Syntax Diagrams*, the user has to specify an initial mesh by joining together blocks of macro elements having the form of a (3D) hexahedron, (3D) tetrahedron or (3D) prism and (2D) rectangle or (2D) triangle, as shown in Fig. 3.1. Macro elements are used to define an initial mesh fine enough to specify material regions, boundary conditions or any other region of interest. For computational purposes this initial mesh may be too coarse, however, hexahedral and rectangular macro elements can be locally and adaptively refined, whereas tetrahedral, prismatic and triangular macro elements support at the present time just a homogeneous refinement. In particular, each hexahedral/rectangular macro element can be recursively refined

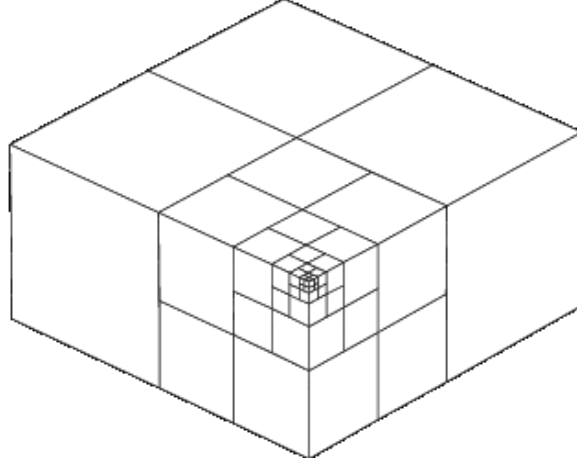


Figure 4.1: An irregular finite element mesh in *SESES* with hanging nodes.

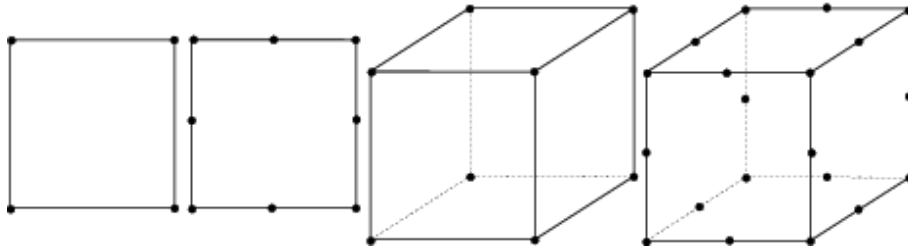


Figure 4.2: Degrees of freedom for  $H^1(\Omega)$  rectangular elements with built-in support for hanging nodes on irregular meshes.

into eight/four sub-elements as long as the neighbors do not have a refinement level larger or smaller than one. This last condition has been imposed in order to avoid an unreasonably complex implementation. This refinement scheme is extremely flexible towards local refinement and also has a compact octree-data representation and a simple implementation, see Fig 4.1. However, these irregular meshes with hanging nodes are a priori not admissible for finite element discretizations and some numerical amendments are required.

A second important task of any finite element method is the specification of the field approximation. Several finite elements of different type and degree are available but the usage of irregular meshes as shown in Fig 4.1 sets additional constraints and may result in complex algorithms so that only few of them are available in *SESES*. Let us assume we have a linear governing equation for the unknown field  $u$  and that a solution of  $u$  is sought either in one of the following Sobolev spaces  $H^1(\Omega)$ ,  $H(\text{div}; \Omega)$  or  $H(\text{curl}; \Omega)$ , henceforth denoted by  $\mathcal{H}$ . A finite element (FE) consists in first defining for each element  $\Omega^e$ , a functional space  $\mathcal{P}^e \subset \mathcal{H}(\Omega^e)$  with a basis  $\phi_j^e(\mathbf{x})$ ,  $j = 1 \dots n^e$  called the element *shape functions*. The FE is said to be *conform* if we can combine and patch together the element shape functions  $\phi_j^e$  into  $n$  global ones  $\phi_i$ ,  $i = 1 \dots n$  so that on each element  $\phi_i|_e$  is either equal to 0 or some  $\phi_j^e$  and globally we have  $\phi_i \in \mathcal{H}$ . This last condition is the most critical one and this is indeed satisfied if for the spaces  $H^1$ ,  $H(\text{div})$  or  $H(\text{curl})$ , the global shape functions  $\phi_i$  between element boundaries are continuous, have a continuous normal or tangential component, [11]. Finally, by proper

weighting the linear governing equation with  $n$  different functions, integrating over  $\Omega$  and using the approximation  $u_h = \sum_i u_j \phi_j \in \mathcal{H}$  with unknowns  $u_j$ , one generally obtains a linear system of equations

$$S\vec{u} = \vec{f}, \quad (4.5)$$

with  $S$  a positive definite matrix to be solved for  $\vec{u}$ . Several conformal finite elements are available to discretize the governing equations on an admissible finite element mesh. However, as soon as we allow irregular meshes with hanging nodes as in Fig 4.1, the construction of a conformal FE space gets more involved. The reason is that on irregular meshes, we have to explicitly enforce the above continuity requirements at the element boundaries. Without considering a particular FE, it is not possible to give all details and so we just say that in order to fulfill these requirements, one can proceed exactly as for regular meshes but at the end some unknowns  $u_j$  become now linear dependent from other ones. Let us therefore use the split  $\vec{u} = (\vec{u}_I, \vec{u}_D)$  with  $\vec{u}_I$  and  $\vec{u}_D$  representing the independent and dependent unknowns. The values of  $\vec{u}_D$  for a conformal approximation  $u_h = \sum_i u_j \phi_j \in \mathcal{H}$  are uniquely determined by the values  $\vec{u}_I$ . In matrix notation we have  $\vec{u}_D = \Lambda \vec{u}_I$  or equivalently  $(\Lambda, -I)\vec{u} = 0$  with  $I$  the identity matrix. The linear system (4.5), written in the form

$$S\vec{u} = \vec{f} \Leftrightarrow \begin{pmatrix} S_{II} & S_{ID} \\ S_{DI} & S_{DD} \end{pmatrix} \begin{pmatrix} \vec{u}_I \\ \vec{u}_D \end{pmatrix} = \begin{pmatrix} \vec{f}_I \\ \vec{f}_D \end{pmatrix}, \quad (4.6)$$

together with the extra constraint equations is now overdetermined. To compute a solution we solve exactly the constraint equations  $(\Lambda, -I)\vec{u} = 0$  and only approximately the equations (4.6) by means of a minimum problem  $\vec{r} = \vec{f} - S\vec{u}$ ,  $\|\vec{r}\|^2 = \min$ , for some properly chosen norm  $\|\cdot\|$ . By the method of Lagrange multipliers we have to solve a saddle point problem for the functional  $\|\vec{r}\|^2 + \vec{\lambda}^T (\Lambda, -I)\vec{u}$ . The condition of vanishing derivative for the variables  $(\vec{u}, \vec{\lambda})$ , together with the norm  $\|\vec{x}\|^2 = 1/2 \vec{x}^T S^{-1} \vec{x}$  yields the system of equations

$$\begin{cases} \vec{r} - (\Lambda, -I)^T \vec{\lambda} = 0, \\ (\Lambda, -I)\vec{u} = 0. \end{cases} \quad (4.7)$$

After elimination of the unknowns  $(\vec{u}_D, \vec{\lambda})$  from (4.7), we obtain the linear system

$$(S_{II} + S_{IR}\Lambda + \Lambda^T S_{RI} + \Lambda^T S_{RR}\Lambda) \vec{u}_I = (\vec{f}_I + \Lambda^T \vec{f}_D). \quad (4.8)$$

The matrix of (4.8) maintains the same properties of the matrix (4.5), it is positive definite and sparse but it is of smaller dimension, since all dependent unknowns  $\vec{u}_D = \Lambda \vec{u}_I$  have been eliminated. The positive definite property is easily seen by rewriting the matrix (4.8) in the form

$$S_{II} + S_{ID}\Lambda + \Lambda^T S_{DI} + \Lambda^T S_{DD}\Lambda = \begin{pmatrix} I \\ \Lambda \end{pmatrix}^T \begin{pmatrix} S_{II} & S_{ID} \\ S_{DI} & S_{DD} \end{pmatrix} \begin{pmatrix} I \\ \Lambda \end{pmatrix}. \quad (4.9)$$

It is interesting to note that the amendments of (4.8) although written out for the global system, can be performed locally at the element level. During the computation of the element contributions, we do not have to differentiate between elements having dependent or independent unknowns. The only modification required is performed before assembling the element contributions into the global system. If such contributions

belong to dependent unknowns, by appropriately defining the element incidences as the incidences of solely independent unknowns, it is possible to quickly reflect the changes given by (4.8) at the element level and the adjusted element stiffness matrix and element residual vector do not even change their size.

In conclusion, we have shown how to construct conformal FE spaces on irregular meshes with hanging nodes. By numbering separately unconstrained nodes, edges and faces of the mesh and by classifying the various mesh constraints, one can provide support for any type of FEs also on irregular meshes. However, for high degree FEs, the bookkeeping necessarily to specify the dependency matrix  $\Lambda$  quickly becomes unpractical so that one is actually limited to first and second order FE methods. At the present time, the *SESES* mesh generator supports numbering and constraint handling for  $H^1(\Omega)$  and  $H(\text{curl}; \Omega)$  rectangular elements of first and 2nd order. For  $H^1(\Omega)$ , they are show in Fig 4.2 with their degrees of freedom being the function values at the marked dot. The 2nd order is a *serendipity* type and it is characterized by not having middle face (3D only) or internal degrees of freedom. For  $H(\text{curl}; \Omega)$ , we have the first family of Nedelec's elements as given in [12]. Other finite elements which can be embedded in the ones supported by the mesh generator may be used as well, as for example the case for beam and shell FEs.

## 4.2 Finite Elements for the Laplace equation

As an example, we shortly present the discretization of the Laplace equation

$$\nabla \cdot (a \nabla u) + f - cu = 0, \quad (4.10)$$

for the scalar field  $u$  with classical conforming finite elements of  $H^1(\Omega)$  where for the sake of simplicity of notation, we only consider homogeneous Dirichlet BCs. Let  $\Omega \subset \mathbb{R}^3$  be a bounded domain with a Lipschitz-continuous boundary  $\partial\Omega$  and let  $\partial\Omega_D \subset \partial\Omega$  with a positive Lebesgue measure. We use the standard notation of Sobolev space  $H^1(\Omega)$ , inner-product  $(\cdot, \cdot)$  in  $L^2(\Omega)$  or  $(L^2(\Omega))^3$  and define the space  $V = \{v \in H^1(\Omega) : v|_{\partial\Omega_D} = 0\}$ . Since the linear trace operator  $tr : H^1(\Omega) \rightarrow L^2(\partial\Omega_D)$  is continuous, the space  $V$  is a proper closed subspace of  $H^1(\Omega)$  and thus a Hilbert space. The weak formulation of (4.10) reads: find  $u \in V$  such that for all  $v \in V$  we have

$$\mathcal{A}(u, v) = \mathcal{F}(v), \quad (4.11)$$

with  $\mathcal{A}(u, v) = (a \nabla u, \nabla v) + (cu, v)$  and  $\mathcal{F}(v) = -(f, v)$ . Formally the solution of (4.11) is equivalent to the solution of (4.10) with the boundary conditions  $u|_{\partial\Omega_D} = 0$  and  $a \nabla u \cdot \mathbf{n}|_{\partial\Omega \setminus \partial\Omega_D} = 0$  where  $\mathbf{n}$  is the unit outer normal vector to the boundary. If the functions  $a, c, f$  are bounded,  $c \geq 0$  and if there exists a constant  $a_0 > 0$  such that  $a \geq a_0$  holds on  $\Omega$  then the problem (4.11) has a unique solution. In fact, the forms  $\mathcal{A}(u, v)$  and  $\mathcal{F}(v)$  are all linear and continuous in their arguments, the form  $\mathcal{A}(u, v)$  is  $V$ -elliptic since  $\mathcal{A}(u, u) \geq a_0(u, u)$ ,  $\forall u \in V$  and therefore by the Lax-Milgram lemma [5], the problem (4.11) has an unique solution. The discretization of the variational form (4.11) is a formal step based on the choice of a finite dimensional subspace  $V_h \subset V$ . Let  $D_h$  be a disjoint element decomposition of  $\Omega$  with  $\overline{\Omega} = \bigcup_{e \in D_h} \overline{\Omega^e}$  and  $\Omega^e \cap \Omega^{e'} = \emptyset$  for



$e \neq e'$ . Then for each element  $\Omega^e$ , we select a simple functional space  $\mathcal{P}^e$  on  $\Omega^e$  of low dimension and consider the subspace

$$V_h = \{\phi \in V : \forall e \in D_h, \phi|_e \in \mathcal{P}^e\}. \quad (4.12)$$

As noted previously, the inclusion  $\phi \in V$  for functions constructed and patched together from the spaces  $\mathcal{P}^e$  is the most difficult part to fulfill. The discretized form of the problem (4.11) is simply given by: find a  $u_h \in V_h$  satisfying

$$\mathcal{A}(u_h, v_h) = \mathcal{F}(v_h), \quad (4.13)$$

for all  $v_h \in V_h$ . The unique solution of the discretized problem (4.13) is inherited from the uniqueness of the weak form (4.11), since  $V_h \subset V$  holds. In terms of matrix notation, once we have selected a basis for the space  $V_h = \{H_0, H_1, \dots\}$ , we can write (4.13) as

$$\vec{R} = S\vec{u} - \vec{f} = 0, \quad (4.14)$$

with  $\vec{u} = (u_0, u_1, \dots)$ ,  $S_{ij} = ((a\nabla H_i, \nabla H_j) + (cH_i, H_j))$ ,  $f_i = (f, H_i)$  and the field approximation given by  $u_h = \sum_i H_i u_i$ . The matrix  $S$  is positive-definite and with the choice of a basis  $\{H_0, H_1, \dots\}$  with local support [5] has a sparse character.

## Mixed Finite Elements

An alternative way to solve the Laplace equation (4.10) is to start with the following equivalent system of first order PDEs in the variables  $(\mathbf{p}, u)$

$$\begin{aligned} \mathbf{p} &= a\nabla u, \\ \nabla \cdot \mathbf{p} + f - cu &= 0. \end{aligned} \quad (4.15)$$

The weak formulation of (4.15) is as follow. For  $V = \{v \in H^1(\Omega) : v|_{\partial\Omega_D} = 0\}$  and  $W = L^2(\Omega)$ , find a pair  $(\mathbf{p}, u) \in W \times V$  such that for all  $(\mathbf{q}, v) \in W \times V$  we have

$$\begin{aligned} \mathcal{A}(\mathbf{p}, \mathbf{q}) + \mathcal{B}(\mathbf{q}, u) &= 0, \\ \mathcal{B}(\mathbf{p}, v) + \mathcal{D}(u, v) &= \mathcal{F}(v), \end{aligned} \quad (4.16)$$

with  $\mathcal{A}(\mathbf{p}, \mathbf{q}) = (a^{-1}\mathbf{p}, \mathbf{q})$ ,  $\mathcal{B}(\mathbf{q}, v) = -(\mathbf{q}, \nabla v)$ ,  $\mathcal{D}(u, v) = -(cu, v)$  and  $\mathcal{F}(v) = -(f, v)$ . Formally the solution of (4.16) is equivalent to the solution of (4.15) with the boundary conditions  $u|_{\partial\Omega_D} = 0$  and  $\mathbf{p} \cdot \mathbf{n}|_{\partial\Omega \setminus \partial\Omega_D} = 0$  where  $\mathbf{n}$  is the unit outer normal vector to the boundary. If the functions  $a, c, f$  are bounded,  $c \geq 0$  and if there exists a constant  $a_0 > 0$  such that  $a_0 < a$  holds, then the problem (4.16) has a unique solution. Let us recall briefly the proof. The forms  $\mathcal{A}(\mathbf{p}, \mathbf{q})$ ,  $\mathcal{B}(\mathbf{q}, v)$ ,  $\mathcal{D}(u, v)$  and  $\mathcal{F}(v)$  are all linear and continuous in their arguments. Further, the form  $\mathcal{A}(\mathbf{p}, \mathbf{q})$  is  $W$ -elliptic since  $\mathcal{A}(\mathbf{q}, \mathbf{q}) \geq a_0(\mathbf{q}, \mathbf{q})$ ,  $\forall \mathbf{q} \in W$  and the form  $\mathcal{D}(u, v)$  is negative. As last we have to prove the inf-sup condition for the form  $\mathcal{B}(\mathbf{q}, v)$

$$\beta = \inf_{\{v \in V : \|v\|_V = 1\}} \sup_{\{\mathbf{q} \in W : \|\mathbf{q}\|_W = 1\}} \mathcal{B}(\mathbf{q}, v) > 0. \quad (4.17)$$

Since  $\partial\Omega_D$  has positive Lebesgue measure, the  $H^1(\Omega)$ -norm is equivalent to the semi-norm  $|\cdot|_1 = (\nabla \cdot, \nabla \cdot)^{1/2}$ , [5]. For  $|v|_1 = 1$ , we have  $\mathcal{B}(-\nabla v, v) = |v|_1 = 1$  and so the above

inf-sup condition holds with  $\beta = 1$ . We can now apply the abstract theory developed in [15] to finish the proof. The discretization of the variational form (4.16) is a formal step based on the choice of finite dimensional subspaces  $W_h \subset W$ ,  $V_h \subset V$ , however, here the inf-sup condition is not automatically inherited and need to be proven for each choice of the subspaces. Let's consider the subspaces

$$\begin{aligned} V_h &= \{\phi \in V : \forall E \in D_h, \phi|_E \in \mathcal{P}^e\}, \\ W_h &= \{\mathbf{q} \in W : \forall E \in D_h, \mathbf{q}|_E \in \nabla \mathcal{P}^e\}. \end{aligned} \quad (4.18)$$

with  $\mathcal{P}^e$  a finite element family as used in (4.12). The discretized form of the problem (4.16) is given by: find a pair  $(\mathbf{p}_h, u_h) \in W_h \times V_h$  satisfying the system

$$\begin{aligned} \mathcal{A}(\mathbf{p}_h, \mathbf{q}_h) + \mathcal{B}(\mathbf{q}_h, u_h) &= 0, \\ \mathcal{B}(\mathbf{p}_h, v_h) + \mathcal{D}(u_h, v_h) &= \mathcal{F}(v_h), \end{aligned} \quad (4.19)$$

for all  $(\mathbf{q}_h, v_h) \in W_h \times V_h$ . By our choice of have  $W_h$ , we have  $\nabla V_h \subset W_h$  so that the inf-sup condition holds again with  $\beta = 1$  and the solution of (4.19) is unique. In terms of matrix notation, once we have selected bases for the spaces  $W_h = \{\mathbf{C}_0, \mathbf{C}_1, \dots\}$  and  $V_h = \{H_0, H_1, \dots\}$ , we can write (4.19) as

$$\begin{pmatrix} A & B \\ B^T & -C \end{pmatrix} \begin{pmatrix} \vec{p} \\ -\vec{u} \end{pmatrix} = \begin{pmatrix} 0 \\ \vec{f} \end{pmatrix}, \quad (4.20)$$

with  $\vec{u} = (u_0, u_1, \dots)$ ,  $\vec{p} = (p_0, p_1, \dots)$ ,  $A_{ij} = (a^{-1}\mathbf{C}_i, \mathbf{C}_j)$ ,  $B_{ij} = (\mathbf{C}_i, \nabla H_j)$ ,  $C_{ij} = (cH_i, H_j)$ ,  $f_i = (f, H_i)$  and the field approximations  $\mathbf{p}_h = \sum_i \mathbf{C}_i p_i$ ,  $u_h = \sum_i H_i u_i$ . The choice of the space  $W_h \subset L^2(\Omega)$  frees us from enforcing a continuity requirement for  $\mathbf{p}_h$  and therefore we can easily choose a basis where the matrix  $A$  assumes a block-diagonal form. Instead of directly inverting the system (4.20) we can easily compute the inverse matrix  $A^{-1}$  by inverting each single diagonal block of small size and then the linear system

$$\vec{R} = (B^T A^{-1} B + C)\vec{u} - \vec{f} = 0, \quad (4.21)$$

with  $(B^T A^{-1} B + C)$  a positive-definite matrix.

## Current Conservation

Although finite element methods start with a weak formulation and so differently from finite volume methods do not have a strong form of flux conservation built-in at the element level, they indeed have a strong conservation form for the total current. From the divergence form  $\nabla \cdot \mathbf{p} + f - cu = 0$ , by applying Gauss's theorem with  $\mathbf{p} \cdot \mathbf{n}|_{\partial\Omega \setminus \partial\Omega_D} = 0$ , we obtain  $I_c = \int_{\partial\Omega_D} \mathbf{p} \cdot \mathbf{n} d\gamma = (cu, 1) - (f, 1)$ . This property is also inherited by the FE discretization, if the in-out flow of current  $I_c$  is computed as follows

$$I_c = \sum_{\forall \text{Node } i \text{ on } \partial\Omega_D} \{(\mathbf{p}_h, \nabla H_i^*) + (cu_h, H_i^*) - (f, H_i^*)\}, \quad (4.22)$$

and the terms  $(cu_h, H_i^*)$ ,  $(f, H_i^*)$  are integrated exactly. With (4.13) and the property  $\sum_i H_i^* = 1$  we obtain

$$I_c = \sum_i \{(\mathbf{p}_h, \nabla H_i^*) + (cu_h, H_i^*) - (f, H_i^*)\} = (cu_h, 1) - (f, 1). \quad (4.23)$$



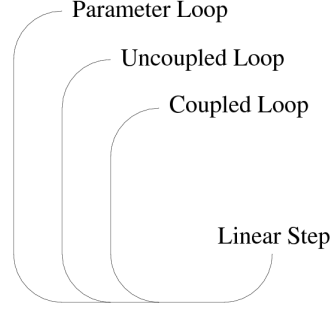


Figure 4.3: Non-linear iteration loops.

### 4.3 Non-Linear Solution Algorithms

In this section we present algorithms for computing non-linear stationary solutions. For this task, there is the possibility to devise an iterative algorithm based on three different nested loops: parameter, uncoupled and coupled loop and where at the innermost location a linear system of equations is solved, as sketched in Fig. 4.3. The coupled and uncoupled loops are used to compute single solutions, whereas the parameter loop is used to compute a family or path of solutions with respect to a user parameter. The parameter can be a global variable or the system defined simulation time. Since we are solving stationary problems, the time is a pseudo time that has to be interpreted like any other user parameter, see also the Section 3.2 *User Parameters*. Computing a family of solutions presents many interesting aspects and will be first discussed in the Section 4.4 *Continuation Methods*, here we present the tools to compute single solutions.

Assume we are looking for steady state solutions of a system of  $n$  PDEs for the dof-fields  $(u^1, \dots, u^n)$ . The spatial discretization of each equation will introduce a vector of dof-values  $\vec{u}^i \in \mathbb{R}^{m_i}$  and  $m_i$  associated residual equations  $\vec{R}^i = 0$  to be solved. Since the governing equations are assumed to be coupled, we generally have  $\vec{R}^i = \vec{R}^i(\vec{u})$  with  $\vec{u} = (\vec{u}^1, \dots, \vec{u}^n)$  and by collecting all residual equations  $\vec{R} = (\vec{R}^1, \dots, \vec{R}^n)$ , we are left to solve a system of non-linear equations  $\vec{R}(\vec{u}) = 0$  of dimension  $N = \sum_{i=1}^n m_i$ . In general, there is not a ready made solution to solve these non-linear equations, also because solutions need not to exist or be unique. Practically all methods are of an iterative nature, they use a starting approximative solution  $\vec{R}(\vec{u}_0) \approx 0$  and at each step  $k \geq 0$ , they generate better approximations  $\vec{u}_{k+1}$  hopefully with  $\vec{R}(\vec{u}_\infty) = 0$ . The methods of interest to us are based on Newton's algorithm or on some modification thereof, see for example [13]. At each step, the method consists in performing a Taylor expansion of  $\vec{R}(\vec{u})$  at  $\vec{u}_k$ , discarding all terms of second and higher order

$$\vec{R}(\vec{u}) \approx \vec{R}(\vec{u}_k) + D_u R(\vec{u}_k)(\vec{u} - \vec{u}_k) \quad \text{with} \quad D_u R = \frac{\partial \vec{R}}{\partial \vec{u}}, \quad (4.24)$$

and solving the remaining linear system for  $\vec{u}$  to get the new approximation  $\vec{u}_{k+1}$  for the solution

$$\vec{u}_{k+1} = \vec{u}_k - D_u R(\vec{u}_k)^{-1} \vec{R}(\vec{u}_k). \quad (4.25)$$

Various convergence criteria are available to stop the iteration (4.25), most of them based on the norm of the increments  $\Delta \vec{u}_{k+1} = \vec{u}_{k+1} - \vec{u}_k$  or the residuals  $\vec{R}(\vec{u}_k)$ .

There is also a large variety of algorithms considering modified increments in order to improve and to speed-up the convergence. Newton's algorithm (4.25) is a full coupled algorithm, since the solution updates are computed for all fields together. This method has the drawback that the linear systems to be solved are almost always non-symmetric and of large size  $N = \sum_{i=1}^n m_i$ . To reduce the large memory requirements of Newton's algorithm, a common practice is to use the following approximation for the Jacobi matrix

$$D_u R \approx \begin{pmatrix} \frac{\partial \vec{R}^1}{\partial \vec{u}^1} & 0 & \dots \\ 0 & \ddots & 0 \\ \vdots & 0 & \frac{\partial \vec{R}^n}{\partial \vec{u}^n} \end{pmatrix}. \quad (4.26)$$

Because (4.26) has now a block diagonal form, we can solve each block separately. This uncoupled algorithm, known as the Gauss-Seidel algorithm, simply solves each discretized PDE iteratively one after the other, until the residuals equations  $\vec{R}(\vec{u})$  have converged.

In general to solve PDEs with weak non-linearities, the uncoupled algorithm is the method of choice, whereas the coupled method is preferred for highly non-linear equations. However, there are counterexamples, as for the highly non-linear PDEs of the semiconductors drift-diffusion model. Here Newton's algorithm is not always the best choice and the Gauss-Seidel algorithm may perform better. However, this strongly depends on the type of the solution being computed. For multiple fields and highly non-linear problems, the coupled and uncoupled algorithm alone may not be enough to compute the solution. For example, the coupled algorithm may diverge abruptly and the uncoupled algorithm may oscillate indefinitely. The problem here is that the starting solution is a bad approximation of the final solution. To compute the final solution, we have thus to compute a better initial guess; to this end, the parameter iteration loop can be used as described in the Section 4.4 *Continuation Methods*

In *SESES* there is not only the possibility to solve a set of non-linear PDEs with the coupled Newton's algorithm or the uncoupled Gauss-Seidel algorithm but a more general approach is available. To this end we have introduced the concept of a block as a set of PDEs to be solved. For example, if for a set of equations we define the same number of blocks and for each block a single but different equation, then we have the uncoupled or Gauss-Seidel algorithm. One block containing all the equations is equivalent to the coupled Newton's algorithm. A block structure is defined with a `BlockStruct` statement explained on p. 78. Moreover, the block structure can be changed during a solution algorithm, if necessary, for example, to speed-up convergence or to achieve convergence at all. When a block structure has been defined, the overall solution strategy consists in sequentially computing block solutions and repeating the cycle, this is the uncoupled or Gauss-Seidel loop. Block solutions are then computed with the Newton's algorithm, this is the coupled loop nested inside the uncoupled one. The convergence of the coupled loop is governed by the norms and a user specified criterion evaluated at the end of each linear solution step. If the block defines only linear equations, no convergence criteria need to be specified and a single linear solution step will be done. The convergence of the uncoupled loop is similarly defined by norms and a user specified criterion with the major difference, however, that the norms are the ones computed in the first linear solution step when solving for the blocks. The default strategy for the uncoupled loop is to stop after the

first cycle.

The coupled loop converges quadratically if the approximate solution slightly differs from the sought one, which corresponds to the best available convergence behavior. However, starting solutions are never so nearly exact and methods must be devised, not so much to speed-up the convergence but, principally to avoid divergence of Newton's algorithm outside its quadratic convergence radius. In this region the computed solution increments are generally overestimated and uncorrelated among different solution steps so that a simple limitation of the increments before updating the solution is the most widely used method to circumvent divergence. The fast quadratic convergence of the algorithm close to the solution is counterbalanced by the need to solve a linear system at each iteration. If a direct solver is used, however, it is possible to reuse the previously factorized matrix. The approximate tangent matrix will slow down the convergence behavior but the expensive numerical factorization is avoided and the overall expenditure can be improved. If an iterative solver is used, there is no need to fully solve the linear system at each iteration, one just have to solve enough to guarantee a good linear convergence behavior. Non-default increments to improve and speed-up the convergence of Newton's algorithm are defined with the `Increment` statement explained on p. 71.

The uncoupled loop behaves completely different from the coupled loop, its convergence being only linear and it is not uncommon to have very slow convergence, i.e. factors near to one. In this case the sum of the increments computed in the inner coupled loop (block increments) is generally underestimated and methods can be devised to accelerate the convergence. An extrapolation technique requiring the storage of previously computed block increments is generally used but its success depends essentially on the solution being computed. Once the storage of block increments has been supplied, various extrapolation techniques can be implemented and experimented with to select the most suitable one. If we assume the block increment  $\Delta \vec{u}_k$  at iteration  $k$  being correlated to the previous one by the scaling factor  $\lambda (\lambda < 1)$

$$\Delta \vec{u}_k = \lambda \Delta \vec{u}_{k-1}, \quad (4.27)$$

using the limit of the geometrical series, we can predict the sum of all the next block increments

$$\Delta \vec{u}_{N,\infty} = \sum_{k=N}^{\infty} \Delta \vec{u}_k = \frac{\Delta \vec{u}_k}{1 - \lambda}, \quad (4.28)$$

and use  $\Delta \vec{u}_{N,\infty}$  to update the solution instead of  $\Delta \vec{u}_N$ . The success of the method strongly depends on the validity of the assumption (4.27) and the choice of the parameter which can be evaluated vector-wise or component-wise. Extensions of this method for multiple block increments are also available. At the present time, however, block increment extrapolation has been disabled.

## 4.4 Continuation Methods

When performing numerical modeling one is often interested in computing families of solutions depending on some input parameters. In *SESES* these parameters are called

user parameters and are presented in the Section [3.2 User Parameters](#). The parameter iteration loop defined around the uncoupled and coupled loop, see Fig. [4.3](#), is then used to change a user parameter and to solve at each step the governing equations. This is done with the `Solve Stationary` statement explained on p. [80](#). Since almost any input value can depend on a user parameter, a large variety of solution families can be computed. In the following, we will consider a user parameter  $\lambda$  in the range  $\lambda \in [0, 1]$ . After proper discretization of the governing equations, one is left with finding a family or path of solutions  $\vec{u}(\lambda)$  for the equations  $\vec{R}(\vec{u}, \lambda) = 0$  with  $\vec{R}, \vec{u} \in \mathbb{R}^n$ . We assume that for  $\lambda = 0$  the solution to  $\vec{R}(\vec{u}, 0) = 0$  is known and given by  $\vec{u}(0)$ . Different scenarios are possible here.

- One is interested in all solutions for  $\lambda \in [0, 1]$  and one knows that solutions exist and are unique.
- One is only interested in a single solution  $\vec{R}_1(\vec{u}_1) = 0$  but a direct solution fails or is too slow. Assuming we know a solution  $\vec{u}_0$  to a more simple system of equations  $\vec{R}_0(\vec{u}_0) = 0$ , one can introduce a smooth  $\lambda$  dependency in the problem and define a new system  $\vec{R}(\vec{u}, \lambda)$  with the properties  $\vec{R}_0(\vec{u}) = \vec{R}(\vec{u}, 0)$  and  $\vec{R}_1(\vec{u}) = \vec{R}(\vec{u}, 1)$ . By iteratively increasing the value of  $\lambda$  from 0 to 1 and by computing the solutions of  $\vec{R}(\vec{u}, \lambda) = 0$ , one ends up with the solution of the algebraic system of equations  $\vec{R}_1(\vec{u}_1) = 0$ . The variation  $\lambda$  from one solution step to the next one should be small enough so that the uncoupled or coupled solution algorithms can be used to compute the intermediary solutions. This case is very similar to the previous one, the only difference is that one is interested in computing the final solution as fast as possible. The success of this method clearly depends on the choice of the  $\lambda$  dependency for  $\vec{R}(\vec{u}, \lambda)$ .
- The path of solutions  $\vec{u} = \vec{u}(\lambda)$  is a complex function of  $\lambda$  and for example solutions may not need to exist or be unique. Therefore one would like to know valid values for  $\lambda$  and be able to compute non unique solutions.

Let us start with the most simple situation and let us assume that for a given  $\lambda_0$  a solution  $\vec{u}_0$  of  $\vec{R}(\vec{u}, \lambda_0) = 0$  exists with  $D_u R = \partial \vec{R} / \partial \vec{u}$  non-singular. With some smoothness assumptions on  $\vec{R}(\vec{u}, \lambda)$ , the implicit function theorem assures the existence of a smooth family of solutions  $\vec{u}(\lambda)$  for  $\lambda$  in a neighborhood of  $\lambda_0$ . If when increasing  $\lambda$  the norm of  $(D_u R)^{-1}$  remains bounded, then one obtains the existence of the solution for  $\lambda = 1$ . Thus for this case, computing path of solutions does not really differs from computing single solutions, one has just to specify the increments  $\Delta\lambda$  for the parameter loop. However, a proper and optimized step selection may not always be an easy task so that automatic algorithms are welcome as for example try and error algorithms based on convergence failure or similar criteria. One can also speed-up the whole solution process, if after incrementing  $\lambda$ , one first tries to predict the solution before solving and thus correcting the equations.

A simple predictor approach uses extrapolation with respect to  $\lambda$  but requires the storage and bookkeeping of previously computed solutions. In *SESES*, polynomial or Lagrangian extrapolation up to the fifth order and rational extrapolation up to the third order for the numerator and denominator polynomial is implemented. They

are selected with a `Extrapolation` statement explained on p. 69. High degree Lagrangian extrapolation is generally not so useful because it tends to be highly divergent and in this case rational extrapolation is better suited, see [16]. Let  $f_i = f(\lambda_i)$ ,  $i = 0 \dots n$  be the solution values at a given mesh node for the user parameter values  $\lambda_i$ ,  $i = 0 \dots n$ . For the Lagrangian extrapolation, the mesh node solution at the value  $\lambda_{n+1}$  is taken to be  $f(\lambda_{n+1}) = P_n(\lambda_{n+1})$ , where  $P_n(\lambda)$  is a polynomial of maximal degree  $n$  satisfying  $P_n(\lambda_i) = f_i$ ,  $i = 0 \dots n$ . For a rational extrapolation, we use instead  $f(\lambda_{n+1}) = P_n(\lambda_{n+1})/Q_m(\lambda_{n+1})$ ,  $l + m - 1 = n$  where  $P_l$  and  $Q_m$  are polynomials of maximal degree  $l, m$  satisfying  $f_i = P_l(\lambda_i)/Q_m(\lambda_i)$ ,  $i = 0 \dots n$ . For a given number of sample points  $n + 1$ , there are of course many choices of the polynomial degrees  $l$  and  $m$ , a good choice is given by  $l = n \operatorname{div} 2$ ,  $m = (n + 1) \operatorname{div} 2$ . The Lagrangian extrapolation  $f(\lambda_{n+1}) = P_n(\lambda_{n+1})$  is a linear function of the values  $f_i = f(\lambda_i)$ ,  $i = 0 \dots n$  and therefore the linear map must be computed only once for all mesh nodes. This is not the case for a rational extrapolation, however, a fast recursive formula is available for its evaluation, see [16].

Another predictor is the ODE Euler approach which computes or approximates  $D_\lambda \vec{u} = \partial \vec{u} / \partial \lambda$  and uses  $\vec{u}_0 + \Delta \lambda D_\lambda \vec{u}$  as predictor. Differentiation of  $\vec{R}(\vec{u}, \lambda)$  with respect to  $\lambda$  yields the relation  $D_\lambda \vec{u} = -(D_u R)^{-1} D_\lambda \vec{R}$ . This method, however, is only useful if the inverse  $(D_u R)^{-1}$  is readily available from a previous solution step. The Euler approach does not seem superior to the chord approach that uses the predictor  $\vec{u}_0 - (D_u R(\lambda_0))^{-1} \vec{R}(\vec{u}_0, \lambda)$  i.e. a first Newton step without updating the tangent stiffness matrix.

On our way to compute the final solution at  $\lambda = 1$ , the matrix  $D_u R$  may become singular, say at  $\lambda_0$ , so that the simple approach of incrementing  $\lambda$  can break down. At these singular points, two cases are distinguished: (1) in a neighbor of  $\lambda_0$  solutions exist or (2) do not. In the first case, incrementing  $\lambda$  may work since we can jump over the singularity, however, in doing so we miss the characterization of the solution at these singular points called bifurcation points. Here the path of solutions may split in several branches. In the second case, the procedure breaks down since no solutions can be found close to  $\lambda_0$  and so  $\lambda$  cannot be incremented. These singular points are called limit or turning points. If at a singular point the determinant  $\det(D_u R)$  changes sign, then from a region with stable solutions, one may pass over to an unstable region requiring a dynamic simulation. Typical examples are the snap-through or snap-back of mechanical structures like curved shells under increasing pressure. If a path bifurcation occurs, then the new paths may be stable or unstable and in general the stable path taken after bifurcation depends on small perturbations of the system. A general analysis of singular points is a complex matter, however, for many problems of practical interest, the singularity points are found to be simple i.e. the matrix  $D_u R$  has rank  $n - 1$  which greatly simplifies the analysis. This is the only case that we consider here and for which numerical algorithms have been implemented.

At limits or turning points, the velocity tends to infinity  $D_\lambda \vec{u} \rightarrow \infty$  and our representation of  $\vec{u}(\lambda)$  breaks down as well as our continuation method since variations of  $\lambda$  and  $\vec{u}$  are becoming perpendicular. By considering the enlarged space  $\mathbb{R}^{n+1}$ , the solutions path  $(\vec{u}, \lambda)$  is a well defined smooth curve which can be parameterized e.g. by  $s$ . Any smooth one-to-one function can be used to change the parameterization, so that one generally picks up the most useful one for numerical computations. Particular



choices will be presented later on and each choice is characterized by an equation of the form  $f(\vec{u}, \lambda, s) = 0$ . As result, in order to compute and pass over turning points, after selection of a parameterization  $s$ , one solves the augmented governing equations  $\vec{G}(\vec{u}, \lambda) = (\vec{R}(\vec{u}, \lambda), f(\vec{u}, \lambda, s)) = 0$  for the unknowns  $\vec{x} = (\vec{u}, \lambda)$ . The solution  $\vec{x}(s)$  now depends from the parameter  $s$  and a family of solutions can be computed as before with the role of  $\lambda$  replaced by  $s$ . The continuation method will work if the new tangent stiffness matrix

$$D_x G = \begin{pmatrix} D_u R & D_\lambda R \\ (D_u f)^T & D_\lambda f \end{pmatrix}, \quad (4.29)$$

is not singular. Since  $\text{rank}(D_u R) = n - 1$ , this will be the case if we have  $D_\lambda R \notin \text{range}(D_u R)$  and  $D_u f \notin \text{range}(D_u R^T)$ . The first condition characterizes turning points while the second one holds by a proper choice of the parameterization for  $s$ . Since the matrix  $D_x G$  is regular, a Newton iteration  $D_x G(\vec{x}_k) \Delta \vec{x} = -\vec{G}(\vec{x}_k)$  with  $\Delta \vec{x} = \vec{x}_{k+1} - \vec{x}_k$  may seem an adequate method to compute a solution. However, the method requires the full tangent matrix  $D_x G$  but unless special considerations are devoted to the choice of the parametric equation  $f(\vec{u}, \lambda, s) = 0$ , the matrix  $D_x G$  loses its sparse character and symmetry. If a direct solver is used, some sort of pivoting should be selected to avoid cancellation. The method is working, but we may also look for methods extending the underlying equation solver that do not impose limitations or requires major modifications. To this end we factorize the matrix  $D_x G$  as follows

$$D_x G = \begin{pmatrix} D_u R & 0 \\ (D_u f)^T & 1 \end{pmatrix} \begin{pmatrix} \text{Id} & \vec{v} \\ 0 & \alpha \end{pmatrix}, \quad (4.30)$$

with  $\vec{v} = (D_u R)^{-1} D_\lambda R$  and  $\alpha = D_\lambda f - (D_u f)^T \vec{v}$ . The solution of  $\vec{G}(\vec{u}, \lambda) = 0$  using the Newton's method can be written by considering the inverse matrix

$$(D_x G)^{-1} = \begin{pmatrix} \text{Id} & -\vec{v}/\alpha \\ 0 & 1/\alpha \end{pmatrix} \begin{pmatrix} (D_u R)^{-1} & 0 \\ -(D_u f)^T (D_u R)^{-1} & 1 \end{pmatrix}, \quad (4.31)$$

in the following form

$$\begin{aligned} (1) \quad & \vec{v} = (D_u R)^{-1} D_\lambda \vec{R}, \quad \vec{w} = (D_u R)^{-1} \vec{R}, \\ (2) \quad & \Delta \lambda = -(f - D_u f^T \vec{w})/\alpha, \\ (3) \quad & \Delta \vec{u} = -\vec{w} - \Delta \lambda \vec{v}. \end{aligned} \quad (4.32)$$

Steps (2) and (3) involve fast numerical operations to be done after step (1) requiring the solutions of two linear systems with the same matrix but different right-hand-sides. If a direct solver is used, this corresponds more or less to the solution of a single linear system. To a first view, we have gained nothing since at the turning point, the matrix  $D_u R$  is singular and this method breaks down as did the original unamended continuation method based on the parameter  $\lambda$ . Here, however, the velocity  $D_s x$  remains bounded and so we have stabilized the system and we can pass over turning points. This in turn let us to choose proportionally larger steps for  $s$  than for  $\lambda$ . Whenever the linearized form (4.32) of the parametric equation  $f(\vec{u}, \lambda, s) = 0$  is used, it is important to correctly predict either the solution  $\vec{u}$  and/or the load parameter  $\lambda$  in order to proceed further along the solution path  $\vec{x}(s)$ . Otherwise whenever reaching turning points, the algorithm tends to *go back on its track* and oscillates indefinitely.

We now turn to the discussion of choosing the parametric equation  $f(\vec{u}, \lambda, s) = 0$  where in general the parameter  $s$  is close related to the length of the curve  $\vec{x}(s)$ . These stabilized methods for passing turning points are therefore called arc-length continuation methods. A practical and therefore popular method is given by looking for the next solution  $\vec{x}$  and by constraining the length of the increments  $\vec{x} - \vec{x}_s$  to a given value  $\Delta s$  with  $\vec{x}_s = (\vec{u}_s, \lambda_s)$  the actual known solution at the actual value of  $s$ . More precisely we have

$$2f(\vec{u}, \lambda, s) = (\vec{u} - \vec{u}_s)^2 \zeta_u^2 + (\lambda - \lambda_s)^2 \zeta_\lambda^2 - \Delta s^2 = 0, \quad (4.33)$$

with  $\zeta_\lambda$  and  $\zeta_u$  scaling factors defining the norm of  $\vec{x} = (\vec{u}, \lambda)$  and  $\Delta s$  the increment of the parameter  $s$ . This form of parametric equation is called spherical arc-length method and by choosing  $\zeta_\lambda = 0$  we have a cylindrical arc-length method [6]. The derivatives used to solve (4.32) are given by  $D_\lambda f = (\lambda - \lambda_s) \zeta_\lambda^2$  and  $D_u f = (\vec{u} - \vec{u}_s) \zeta_u^2$ . The parametric equation (4.33) being a simple function can be directly enforced instead of being linearized as done within (4.32) with the advantages that the equations  $\vec{G} = 0$  are less non-linear and we have more control on following the correct solution path  $\vec{x}(s)$ . To solve for  $\vec{G} = (\vec{R}, f) = 0$ , we therefore linearize only the  $\vec{R}$  term obtaining the relation

$$\begin{cases} \Delta \vec{u} = -(D_u R)^{-1} \vec{R} - (D_u R)^{-1} D_\lambda \vec{R} \Delta \lambda, \\ f(\vec{u}, \lambda, s) = 0. \end{cases} \quad (4.34)$$

The value of lambda  $\Delta \lambda$  is given by solving the quadratic equation  $f(\vec{u} + \Delta \vec{u}, \lambda + \Delta \lambda, s) = 0$  with the value of  $\Delta \vec{u}$  taken from (4.34) and by selecting the proper solution of the two available ones. Here, one chooses the one going in the direction of the velocity  $D_s \vec{x}$ , whereas for the first step one has to decide in which direction to go.

The continuation method using the augmented parametric equation  $f(\vec{u}, \lambda, s) = 0$  to define the new solution parameter  $s$  although more stable with respect to the simple continuation method involving  $\lambda$  may break down at bifurcation points where per definition we have  $D_\lambda R \in \text{range}(D_u R)$  and so the matrix  $D_x G$  becomes singular. Differently from turning points where the solution  $\vec{x}(s)$  is a function of  $s$ , at bifurcation points the path may split in two branches. More branches are not possible from our assumption that singular points must be simple. In order to proceed after a bifurcation point on a branch of our choices, we have first to compute the solution at the bifurcation in order to characterize it and then take further steps. We may of course try to jump over the bifurcation as done for turning points, but nothing will tell us what branches we are following. In order to pass bifurcation points, it is thus essential to compute the solution there. There are algorithms based on bisection that are looking at the determinant  $\det(D_x R)$  in order to locate the singularity, but Newton methods converging quadratically should be preferred. These methods are obtained by extending the system with an equation characterizing the singularity. Common choices for the additional equation are  $\det(D_u R) = 0$  or  $(D_u R)\Phi = 0, \Phi \neq 0$ . The first choice is not well suited for systems arising from a finite element discretization and although the second choice requires the computation of the null mode  $\Phi$  of  $D_u R$ , the overall computational cost does not considerably differ from a single solution step. Clearly these methods can also be used to spot turning points. The extended system has isolated solutions at turning points, however, at bifurcation points the derivative is again singular so that the Newton method need be stabilized to achieve quadratic



convergence. Once the solution at the singular point has been computed with the extended system, perturbation of the solution with the null eigenmode allow to proceed on a selected branch with a standard or augmented continuation method.

## 4.5 Unstationary Solutions

This section describes how to compute unstationary solutions with *SESES* when solving parabolic problems, for example, the heat conduction problem or the drift-diffusion model for semiconductors. At the present time, integration methods for problems with a second order time derivative are not available but will be implemented for the characterization of mechanical structures. For this last example and the linear case an eigen-pair analysis is available.

Several time integration algorithms are available; they are described here in detail in order to use them properly. Let consider a generic non-autonomous parabolic problem

$$\partial_t u - L(t, u) = 0, \quad (4.35)$$

for the field  $u$ . In the linear case with  $L(t, u) = A(t)u$ , if the operator  $A(t)$  is smooth enough, it is possible to show the existence of solutions  $u(x, t) \in H^1(\Omega) \times C^1([0, \infty))$ , see [8]. By taking the existence of an evolutionary solution as granted, we now proceed with the discretization of the parabolic problem. As first step, we perform a discretization with respect to the space variable  $x \in \Omega$ , then with respect to time variable  $t \in [T_0, \infty)$ . For the spatial discretization we use a finite element method which yields the following system of Ordinary Differential Equations (ODE)

$$\partial_t \vec{u} + M^{-1} \vec{R}(t, \vec{u}) = 0. \quad (4.36)$$

The vector  $\vec{R}(t, \vec{u})$  is the residual vector obtained by discretizing the elliptic operator  $L(t, u)$ ; for an example see (4.14) or (4.21). The space discretization of the term  $\partial_t u$  leads to term  $M \partial_t \vec{u}$  where  $M$  is the positive definite mass matrix given by

$$M = \int_{\Omega} \vec{H} \vec{H}^T d\mu, \quad (4.37)$$

and as before we use the approximation  $u_h = \vec{H}^T \vec{u}$  with  $\vec{H}$  a basis of  $H_h \subset H^1(\Omega)$ . For a given initial condition  $\vec{u}_0 = \vec{u}(t = T_0)$ , the system (4.36) has a unique differentiable solution  $\vec{u}(t)$  for  $t \in [T_0, T_{max}]$ , if the residual vector  $\vec{R}(t, \vec{u})$  is continuous, Lipschitz continuous with respect to  $\vec{u}$  and  $|\vec{R}(t, \vec{u})|$  is bounded by  $\tilde{A}$  in the domain  $\{(t, \vec{u}) : T_0 < t \leq T_{max}, |\vec{u} - \vec{u}_0| \leq \tilde{A}(T_{max} - T_0)\}$ .

A common property of ODE systems arising from the discretization of parabolic problems, is that they have a *stiff* character. Although there is not an exact mathematical definition of *stiffness*, these ODE systems may be characterized by the fact that implicit methods work much better then explicit ones and that the length of the integration step is constrained by stability and not by accuracy, see [9, 10]. In the following, we present linear implicit multistep methods which include the class of Backward Differentiation Formula (BDF) methods generally used to solve stiff problems. Although

for explicit methods we do not have to solve large linearized system of equations, explicit methods are not considered because due to stability the step length limitation is generally very restrictive.

Let  $\vec{u}_i$  be the known solutions at the time values  $T_i = T_0 + \sum_{j=0}^{i-1} \Delta T_j$  for  $i = 0 \dots n$ . By allowing the mass matrix  $M(t, \vec{u})$  to have the same dependency as the residual vector  $\vec{R}(t, \vec{u})$ , for a generic linear multistep method the solution  $\vec{u}_{n+1}$  for the next time step  $T_{n+1} = T_n + \Delta T_n$  is given by solving the following problem

$$\sum_{k=0}^{N_1} \alpha_{k,n} \vec{u}_{n+1-k} + \sum_{k=0}^{N_2} \beta_{k,n} M^{-1}(T_{n+1-k}, \vec{u}_{n+1-k}) \vec{R}(T_{n+1-k}, \vec{u}_{n+1-k}) = 0. \quad (4.38)$$

Since we allow for time steps of different size, the coefficients  $\alpha_{k,n}, \beta_{k,n}$  characteristic of the integration method are allowed to vary with the time step  $n$ . Let us use the short notation  $\vec{R}_i = \vec{R}(T_i, \vec{u}_i)$ ,  $M_i = M(T_i, \vec{u}_i)$  and rewrite the above system as

$$\frac{\alpha_{0,n}}{\beta_{0,n}} M_{n+1} \vec{u}_{n+1} + \vec{R}_{n+1} = -M_{n+1} \vec{w}_n, \quad (4.39)$$

with the right-hand side vector

$$\vec{w}_n = \frac{1}{\beta_{0,n}} \left( \sum_{k=1}^{N_1} \alpha_{k,n} \vec{u}_{n+1-k} + \sum_{k=1}^{N_2} \beta_{k,n} M_{n+1-k}^{-1} \vec{R}_{n+1-k} \right), \quad (4.40)$$

not depending on the unknown  $\vec{u}_{n+1}$ . In the derivation of (4.39)-(4.40), we have assumed  $\beta_{0,n} \neq 0$ , i.e. we have an implicit method and divided (4.38) by  $\beta_{0,n}$  and multiplied by  $M_{n+1}$ . This is advantageous in the sense that for stationary or unstationary problems, the residual vector  $\vec{R}_{n+1}$  is unchanged and no mass matrix inverse  $M_{n+1}^{-1}$  need to be computed.

If we consider the general case of a non-linear operator  $L(t, u)$ , the equations (4.39) are non-linear in the unknown vector  $\vec{u}_{n+1}$ . Let us use an iterative Newton's algorithm to compute a numerical solution and let  $\vec{u}_{n+1,i}$  be an approximate solution to (4.39). For the initial approximation  $\vec{u}_{n+1,0}$  the most simple initialization is to use the actual solution  $\vec{u}_n$ . However, other initializations are possible and may lead to a faster convergence of Newton's algorithm. For example,  $\vec{u}_{n+1,0}$  may be extrapolated or predicted from previously computed solutions with an explicit integration method. The next approximation  $\vec{u}_{n+1,i+1}$  of the solution  $\vec{u}_{n+1}$  is given by linearizing the system (4.39) at the point  $\vec{u}_{n+1,i}$ . Newton's algorithm always computes the increment vector  $\Delta \vec{u}_{n+1,i} = (\vec{u}_{n+1,i+1} - \vec{u}_{n+1,i})$ , therefore the linearized system is rewritten in the form

$$\left[ \frac{\alpha_{0,n}}{\beta_{0,n}} M_{n+1,i} + \frac{\partial M_{n+1,i}}{\partial \vec{u}_{n+1,i}} \vec{w}_{n,i} + \frac{\partial \vec{R}_{n+1,i}}{\partial \vec{u}_{n+1,i}} \right] \Delta \vec{u}_{n+1,i} = -\vec{R}_{n+1,i} - M_{n+1,i} \vec{w}_{n,i} \quad (4.41)$$

$$\text{with } \vec{w}_{n,i} = \frac{\alpha_{0,n}}{\beta_{0,n}} \vec{u}_{n+1,i} + \vec{w}_n. \quad (4.42)$$

When using Newton's algorithm, we have just to allocate the extra vector  $\vec{w}_{n,i}$  to store the residual correction (4.42). This vector is updated each time after a Newton's step with the contribution  $(\alpha_{0,n}/\beta_{0,n}) \Delta \vec{u}_{n+1,i}$  to obtain the value  $\vec{w}_{n,i+1}$ . At the beginning

Description	SESES name	Type	Order
Implicit Euler	EulerBackward	$\alpha_0 = 1, \alpha_1 = -1, \beta_0 = h$	1
Midpoint Rule	MidpointRule	$\alpha_0 = 1, \alpha_1 = -1, \beta_0 = \beta_1 = h/2$	2
Generalized trapezoidal Rule $1/2 < \theta < 1$	TrapezoidalRule	$\alpha_0 = 1, \alpha_1 = -1, \beta_0 = (1 - \theta)h, \beta_1 = \theta h$	1
Backward Differentiation Formula	BDF2	$\alpha_0 = 3/2, \alpha_1 = -2, \alpha_2 = 1/2, \beta_0 = h$	2
Backward Differentiation Formula	BDF3	$\alpha_0 = 11/6, \alpha_1 = -3, \alpha_2 = 3/2, \alpha_3 = -1/3, \beta_0 = h$	3

Table 4.2: Multistep integration method for a fixed step length  $h$ .

of a new time step, we need to initialize the residual correction  $\vec{w}_{n,0}$  according to (4.42). This is done with the knowledge of the previous solutions  $\vec{u}_{n-k}$ ,  $k = 0, 1, \dots$  and quasi residual vectors  $(M^{-1}\vec{R})_{n-k}$ . The residual vector  $(M^{-1}\vec{R})_{n+1}$  required by the next integration step is obtained from equation (4.39) as

$$(M^{-1}\vec{R})_{n+1} = -\frac{\alpha_{0,n}}{\beta_{0,n}}\vec{u}_{n+1} - \vec{w}_n = -\vec{w}_{n,\infty}, \quad (4.43)$$

with  $\vec{w}_{n,\infty}$  the residual correction (4.42) left at the end of the Newton's iteration. This procedure works except for the first few integration steps where the residual vectors  $(M^{-1}\vec{R})_{n-k}$  cannot be obtained from previously computed solutions. In this case, if the  $\vec{u}_i$  are the supplied initial solutions at the times  $t_i$  with  $i = 0..n-1$ , then the vectors  $(M^{-1}\vec{R})_i = (M^{-1}\vec{R})(t_i, \vec{u}_i)$  must be computed. However, since the evaluation of the inverse mass matrix  $M^{-1}$  is computationally expensive, this initialization procedure is generally avoided; further we also have to provide the start solutions  $\vec{u}_i$  for  $i = 0..n-1$ . Typically, this problem is solved by adapting the multistep method to the number of available solutions. At the beginning we have only the start solution  $\vec{u}_0$  and by using an integration method during the starting phase where the factors  $\beta_{k,n}$ ,  $k \geq 1$  are zero, the residual vectors  $(M^{-1}\vec{R})_{n+1-k}$  are not required.

The multistep integration methods implemented in SESES are given in Table 4.2 where the coefficients of the methods are given for a fixed step length  $h$ . All integration methods are initialized with a step of the implicit Euler method. This initialization does not require the quasi residual vector  $M_0^{-1}\vec{R}_0$  and is also very stable. The next integration steps are then performed with an integration method of the same family but of lower order until enough starting solutions are available. For example, for the BDF3 algorithm, we use EulerBackward and BDF2 for the first two steps, then from the third step on we can use BDF3.

Although multistep methods can be defined of any order, the Dahlquist barrier theorem states that an  $A$ -stable method has a maximal order of two. However, some higher order methods may also work when solving parabolic problems i.e.  $A$ -stability is sometimes not required.

The integration methods of second order are provided with an algorithm for automatic step selection based on approximating the local truncation error  $\vec{e}_{n+1}$ . Different

methods are available, one of them is based on the property

$$\vec{e}_{n+1} = C\Delta T_n^3 \partial_t^{(3)} \vec{u}_n = C\Delta T_n^3 M^{-1} \partial_t^{(2)} \vec{R}_n, \quad (4.44)$$

where  $C$  is the error constant of the method. The second derivative of the residual vector  $\partial_t^{(2)} \vec{R}_n$  is then approximated by divided difference. Another method is based on computing a predicted solution with the explicit second order method of Adams. Since the methods used for computing and predicting the solution have a different error constant, a difference of both solutions yields an estimation of the local truncation error. At the present time in *SESES* the midpoint rule uses the method based on predicting the solution, whereas the BDF2 algorithm uses the method based on divided difference.

When the local truncation error  $\vec{e}_{n+1}$  is given the new time step is selected as

$$\Delta T_{n+1} = \Delta T_n \|\vec{\tau}_{n+1}\|^{-\frac{1}{3}} \text{ with } \tau_{n+1,i} = \frac{e_{n+1,i}}{\epsilon_{\text{rel}}|u_{n+1,i}| + \epsilon_{\text{abs}}}. \quad (4.45)$$

The vector  $\vec{\tau}_{n+1}$  gives a local estimation of the time step stretching factor for each single residual equation. In (4.45), the new time step is then computed as an average of these stretching factors defined through the norm  $\|\vec{\tau}_{n+1}\|$  where the exponent  $-1/3$  represents the second order character of the integration method. For the averaging process typically one uses the norm

$$\|\vec{\tau}_{n+1}\|^2 = \frac{1}{N} \sum_{i=1}^N \tau_{n+1,i}^2. \quad (4.46)$$

## References

- [1] K. J. BATHE, *Finite Element Procedures*, Prentice Hall-International, 1996.
- [2] D. BRAESS, *Finite elements*, Springer-Verlag, 1997.
- [3] S. C. BRENNER, L. R. SCOTT, *The mathematical theory of finite element methods*, Springer-Verlag, 1994.
- [4] F. BREZZI, L. D. MARINI AND P. PIETRA, *Two-dimensional exponential fitting and applications to Drift-Diffusion models*, SIAM J. Numer. Anal., 26, pp. 1343-1355, 1989.
- [5] P. G. CIARLET, *Basic error estimates for elliptic problems*, Handbook of Numerical Analysis by P.G. Ciarlet and J.L Lions, Volume II, North-Holland, 1991.
- [6] M. A. CRISFIELD, *Non-linear finite element analysis of solids and structures*, Volume I, John Wiley&Sons, 1991.
- [7] A. ERN, J-L GUERMOND, *Theory and Practice of Finite Elements*, Springer-Verlag, 2004.
- [8] H. FUJITA, T. SUZUKI, *Evolution problems*, Handbook of Numerical Analysis by P.G. Ciarlet and J.L Lions, Volume II, North-Holland, 1991.

- [9] E. HAIRER, S. P. NORSETT, G. WANNER, *Solving Ordinary Differential Equations I, Nonstiff Problems*, Springer-Verlag, 1993.
- [10] E. HAIRER, G. WANNER, *Solving Ordinary Differential Equations II, Stiff and Differential-Algebraic Problems*, Springer-Verlag, 1996.
- [11] P. MONK, *Finite element methods for Maxwell's equations*, Oxford University Press, 2003.
- [12] J. C. NEDELEC, *Mixed finite elements in  $\mathbb{R}^3$* , Numer. Math. 35, pp. 341-341.
- [13] J. M. ORTEGA, W. C. RHEINBOLDT, *Iterative solution of non-linear equations in several variables*, Academic Press, 1970.
- [14] A. QUARTERONI, A. VALLI, *Numerical approximation of partial differential equations*, Springer-Verlag, 1994.
- [15] J. E. ROBERTS AND J. M. THOMAS, *Mixed and hybrid methods*, Handbook of Numerical Analysis by P.G. Ciarlet and J.L Lions, Volume II, North-Holland 1991.
- [16] J. STOER, R. BULIRSCH, *Introduction to numerical analysis*, Springer-Verlag, 1993.

## Chapter 5

# Physical Models

This chapter describes all physical models available to solve a rich variety of multi-physics problems. Among others, it is possible to model reactive flows together with several species undergoing chemical reactions. The definition of the species together with their kinetics is completely free and it is part of the problem specification. However, for user friendliness many input symbols have been defined to depend from the species names and so these names must be known at the very beginning when parsing the input files. This is done with the `Species` statement presented on p. 47 where up to seven different species can be defined. We will use the symbols  $\alpha$  and  $\langle\alpha\rangle$  as placeholders for the symbol and name of these defined species. In the following, the governing partial differential equations (PDEs) solved for the unknown dof-fields are presented in the Section 5.1 *Governing Equations*. The collection of physical models, physical coupling terms and built-in functions available to specify the material laws associated with the governing equations are then discussed in Section 5.2 *Material laws*. Any description of a device with help of PDEs is only completed after the specification of the boundary conditions for the dof-fields and Section 5.3 *Boundary Conditions* gives a list of them. The vector and tensor quantities inherent to anisotropic material laws are discussed in the Section 5.4 *Tensors and their Representation*.

### 5.1 Governing Equations

This section describes the governing equations available for computing numerical solutions and which are listed in Table 5.1. Here, the equation name not only specifies the governing equation to be solved but also the finite element type used for its discretization and the associated set of dof-fields which are computed for its solution and for which BCs must be specified. For the two most important classes of Dirichlet and Neumann BCs, the value of the dof-field and its associated flux-field on the boundary is defined. For dof-fields with a vector character, we use the symbol  $\langle T1 \rangle$  as a placeholder and short form notation of any one of its component, see Table 3.1. To solve for one or more equations on a domain  $\Omega_{\text{mat}}$  defined by the material `mat`, one has to enable them with the statement `MaterialSpec mat Equation` explained on p. 54 and boundary conditions (BC) on  $\partial\Omega_{\text{mat}}$  for the associated dof-fields are set with

Equation	Dof-field(s)	Dirichlet BC	Neumann BC	References
CompressibleFlow	Velocity.<T1>	$\mathbf{v}$	$(\boldsymbol{\tau} - \alpha \nabla p) \cdot \mathbf{n}$	(5.1), (5.2)
IncompressibleFlow	Pressure	$p$	$(1 - \alpha)\rho \mathbf{v} \cdot \mathbf{n}$	
PorousFlow	Pressure	$p$	$\rho \mathbf{v} \cdot \mathbf{n}$	(5.8)
Transport< $\alpha$ >	< $\alpha$ >	$x_\alpha$	$\mathbf{j}_\alpha \cdot \mathbf{n}$	(5.10)
Convect< $\alpha$ >	< $\alpha$ >	$c_\alpha$	$c_\alpha \mathbf{V} \cdot \mathbf{n}$	(5.10)
ThermalEnergy	Temp	$T$	$\mathbf{F} \cdot \mathbf{n}$	(5.18)
OhmicCurrent	Psi	$\Psi$	$\mathbf{J} \cdot \mathbf{n}$	(5.19)
ElectroStatic	Phi	$\Phi$	$-\mathbf{D} \cdot \mathbf{n}$	(5.21)
MagnetoStatic	MagnPot	$\Theta$	$\mathbf{B} \cdot \mathbf{n}$	(5.24)
EddyCurrent	Afield.<T1> Vint	$\mathbf{A}$ $\int \Psi dt$	$(\mathbf{H} - \mathbf{H}_0) \times \mathbf{n}$	(5.34)
EddyFree	Afield	$\mathbf{A}$	$(\mathbf{H} - \mathbf{H}_0) \times \mathbf{n}$	(5.36)
Elasticity	Disp.<T1>	$\mathbf{u}$	$(\nabla \varphi \mathbf{S}) \cdot \mathbf{n}$	(5.40), (5.43)
ElasticShell	Disp.<T1>	$\mathbf{u}$	$(\nabla \varphi \mathbf{S}) \cdot \mathbf{n}$	(5.40), (5.43)
Poisson	Phi	$\Phi$	$-\mathbf{D} \cdot \mathbf{n}$	(5.46)
UnipolarN	Phi RedN	$\Phi$ $n/n_{in}$	$-\mathbf{D} \cdot \mathbf{n}$ $\mathbf{J}_n \cdot \mathbf{n}$	(5.46)
UnipolarP	Phi RedP	$\Phi$ $p/n_{ip}$	$-\mathbf{D} \cdot \mathbf{n}$ $\mathbf{J}_p \cdot \mathbf{n}$	(5.46)
UnipolarU	Phi SlotU	$\Phi$ $u$	$-\mathbf{D} \cdot \mathbf{n}$ $\mathbf{J}_n \cdot \mathbf{n}$	(5.46)
UnipolarV	Phi SlotV	$\Phi$ $v$	$-\mathbf{D} \cdot \mathbf{n}$ $\mathbf{J}_p \cdot \mathbf{n}$	(5.46)
BipolarNP	Phi RedN RedP	$\Phi$ $n/n_{in}$ $p/n_{ip}$	$-\mathbf{D} \cdot \mathbf{n}$ $\mathbf{J}_n \cdot \mathbf{n}$ $\mathbf{J}_p \cdot \mathbf{n}$	(5.46)
BipolarUV	Phi SlotU SlotV	$\Phi$ $u$ $v$	$-\mathbf{D} \cdot \mathbf{n}$ $\mathbf{J}_n \cdot \mathbf{n}$ $\mathbf{J}_p \cdot \mathbf{n}$	(5.46)

Table 5.1: Equations, associated dof-fields and the values of Dirichlet and Neumann BCs with  $\mathbf{n}$  the outward normal to the boundary. The symbol <T1> stands for the components of a vector as defined by Table 3.1 and < $\alpha$ > for a defined species or convected field.

the statement BC explained on p. 57.

### Equation `CompressibleFlow, IncompressibleFlow[2,P1V2]`

This equation solves the slight compressible or incompressible Navier-Stokes equations for the pressure  $p$  (Dof-field `Pressure`) and the flow velocity  $\mathbf{v}$  (Dof-field `Velocity.<T1>`) in the fluid domain  $\Omega_{\text{fluid}}$ . The transport equations of a fluid can be described by a set of macroscopic balance equations. The conservation of the total mass reads

$$\partial_t \rho + \nabla \cdot (\rho \mathbf{v}) = 0, \quad (5.1)$$

with  $\rho$  the fluid density and  $\mathbf{v}$  its velocity. The conservation of momentum leads to the following set of partial differential equations

$$\partial_t (\rho \mathbf{v}) + \nabla \cdot (\rho \mathbf{v} \otimes \mathbf{v}) = -\nabla p + \nabla \cdot \boldsymbol{\tau} + \mathbf{f}, \quad (5.2)$$



with  $p$  the fluid pressure,  $\tau$  the rank 2 shear stress tensor and  $\mathbf{f}$  the external force density. Since the pressure  $p$  enters the conservation laws only in differential form, we may as well consider and compute  $p$  relative to a constant ambient pressure. This can be advantageous from a numerical point of view since the variations of  $p$  are generally small and the ambient pressure  $p_{\text{amb}}$  must then only be considered together with material laws as for example in (5.71).

For the stationary case and after using the mass conservation (5.1), the momentum equations can be rewritten as

$$-\nabla \cdot \tau + \nabla p + \tilde{\rho}(\mathbf{v} \cdot \nabla)\mathbf{v} = \mathbf{f}, \quad (5.3)$$

with the force density  $\mathbf{f}$  defined by the material parameter `Force` and the additional scaling factor  $\gamma$  for the convective operator  $\tilde{\rho} = \rho\gamma$  defined by the material parameter `NSConvFac`. This factor has been introduced to model e.g. a creeping flow for which  $\gamma = 0$  can be assumed. With the assumption of a Newtonian fluid, the shear stress tensor  $\tau$  is given by

$$\tau = \mu(\nabla \mathbf{v} + (\nabla \mathbf{v})^T) - \frac{2}{3}\mu(\nabla \cdot \mathbf{v})\mathbf{Id}, \quad (5.4)$$

with  $\mathbf{Id}$  the identity matrix and  $\mu$  the dynamic viscosity defined by the material parameter `Viscosity`. Eqs. (5.1) and (5.2) correspond to the compressible Navier-Stokes equations, a system that has to be closed by defining the fluid density  $\rho$  as a function of state variables like pressure and temperature. This material law is defined by the material parameter `Density` and built-in function `IdealGas` is available to define an ideal gas. For the incompressible model, we assume  $\rho = \text{const}$ , hence the mass balance Eq. (5.1) reduces to the incompressibility condition  $\nabla \cdot \mathbf{u} = 0$  and the momentum balance Eq. (5.3) can be rewritten in the form  $-\mu \nabla^2 \mathbf{u} + \nabla p + \tilde{\rho}(\mathbf{u} \cdot \nabla)\mathbf{u} = \mathbf{f}$ .

The incompressible model is solved using the a SUPG-PSPG-grad-div stabilization scheme allowing the free choice for the pressure and velocity approximations which are either both of first order, first/second order for pressure/velocity or both of second order. The stabilization introduces two free parameters  $\delta_a, \delta_b$  whose order of magnitude are determined by a priori error analysis. In particular, let  $\mathbf{u}_h = \sum_{i,m} u_{im} \mathbf{H}_{im}^u(\mathbf{x})$  and  $p_h = \sum_i H_i^p(\mathbf{x}) p_i$  be the velocity and pressure approximations with  $H_i^u, H_i^p$  the scalar shape functions,  $\mathbf{H}_{im}^u = H_i^u(\mathbf{x}) \mathbf{e}_m$  and  $u_{im}, p_i$  the associated unknowns. By considering the strong residuals  $\mathbf{R}_u = -\nu \nabla^2 \mathbf{u} + \nabla p + \tilde{\rho}(\mathbf{u} \cdot \nabla)\mathbf{u} - \mathbf{f}$  and  $R_p = \nabla \cdot \mathbf{u}$ , we then solve the stabilized weighted residuals

$$\begin{cases} r_{im}^u = \langle \nabla \mathbf{H}_{im}^u, \nu \nabla \mathbf{u}_h \rangle - \langle \nabla \mathbf{H}_{im}^u, p_h \rangle (1 - \alpha) + \langle \mathbf{H}_{im}^u, \nabla p_h \rangle \alpha \\ \quad + \langle \mathbf{H}_{im}^u, \tilde{\rho}(\mathbf{u}_h \cdot \nabla) \mathbf{u}_h - \mathbf{f} \rangle \\ \quad + \delta_a \langle (\mathbf{u}_h \cdot \nabla) \nabla \mathbf{H}_{im}^u, \mathbf{R}_{u_h} \rangle + \delta_b \langle \nabla \cdot \mathbf{H}_{im}^u, R_{p_h} \rangle = 0, \\ r_i^p = \rho \langle H_i^p, \nabla \cdot \mathbf{u}_h \rangle (1 - \alpha) - \langle \nabla H_i^p, \mathbf{u}_h \rangle \alpha + \delta_a \langle \nabla H_i^p, \mathbf{R}_{u_h} \rangle = 0, \end{cases} \quad (5.5)$$

with  $\langle \mathbf{a}, \mathbf{b} \rangle = \int_{\Omega} \mathbf{a} \cdot \mathbf{b} \, dV$  and the switcher  $\alpha = 0, 1$  influencing the setting of Neumann BCs and defined by the global parameter `FlowPartInt`. For equal order approximations, we use the stability parameters  $\delta_a = 0.05L^2 / \sqrt{\mu^2 + L^2 \rho^2 \mathbf{v} \cdot \mathbf{v}}$ ,  $\delta_b = 0.2L^2 / \delta_a$  with  $L$  the element's size and for first/second order approximations the values  $\delta_a = 0.05L^2$ ,  $\delta_b = 0.2$ . These values together with the derivatives  $\partial_v \delta_{1,2}$  may be redefined with the material parameter `FlowStab`.

For the compressible model, we allow an additional mass production rate  $\Pi_0$  defined by the material parameter `Rate0`, hence the steady state mass balance Eq. (5.1) is amended to the form  $\nabla \cdot (\rho \mathbf{v}) = \Pi_0$ . This model is solved using the same SUPG-PSPG stabilization scheme for the velocity with  $\alpha = 1$  whereas for the pressure, we use a penalty method by solving the stabilized weighted residuals

$$r_i^p = \delta_3 \langle \nabla H_i^p, \nabla p_h \rangle + \langle \nabla H_i^p, \rho \mathbf{u}_h \rangle = 0. \quad (5.6)$$

The penalty parameter  $\delta_3$  is defined by the material parameter `PressPenalty` and it is conveniently measured with respect to the hydrodynamic time scale  $\tau_{\text{hydro}} = L^2 \rho / \mu$  with  $L$  the typical length scale of the problem at hand. It is recommended to use  $\delta_3 \approx 10^{-6} \cdot \tau_{\text{hydro}}$ , since too large values result in gross violations of the mass balance and to small ones generate wiggles. The penalty stabilization of the pressure is the only one required, hence it is possible to turn-off the residual stabilization by setting  $\delta_a = \delta_b = 0$ .

This problem description has to be completed by the user with BCs for the dof-field `Velocity.<Tl>` and `Pressure` on  $\partial\Omega_{\text{fluid}}$ . For the velocity, a Dirichlet BC sets the value of the velocity  $\mathbf{v}$  and a Neumann BC sets the value of the tractions  $(\boldsymbol{\tau} - \alpha \nabla p) \cdot \mathbf{n}$  with  $\mathbf{n}$  the normal to the boundary. The constant  $\alpha$  is defined by the global parameter `FlowPartInt` and if 0 just the shear stress component  $\boldsymbol{\tau} \cdot \mathbf{n}$  is defined. For the pressure, a Dirichlet BC sets the value of the pressure  $p$  and a Neumann BC sets the value of the mass flow  $(1 - \alpha) \rho \mathbf{v} \cdot \mathbf{n}$  thus requiring  $\alpha = 0$ , see Table 5.1.

## Equation `PorousFlow[2]`

This equation solves the mass flow eq. (5.1) for the pressure  $p$  (Dof-field `Pressure`) for the case of a porous or irrotational flow in the domain  $\Omega_{\text{porous}}$  using either first or 2nd order  $H^1(\Omega_{\text{porous}})$  FEs. For slow fluid velocities, the Darcy's law states that the pressure drop in the material pores is linear with respect to the velocity, proportional to the medium viscosity  $\mu$  and inversely proportional to the average pore cross-section or material permeability  $k$

$$\nabla p = -\frac{\mu}{k} \mathbf{v} \quad \text{or} \quad \mathbf{v} = -\frac{k}{\mu} \nabla p, \quad (5.7)$$

which for  $\mu/k = \text{const}$  results in an irrotational flow  $\nabla \times \mathbf{v} = 0$ . Insertion of (5.7) in the mass balance eq. (5.1) yields the equation

$$\partial_t \rho - \nabla \cdot \rho \frac{k}{\mu} \nabla p = \Pi_0, \quad (5.8)$$

with the permeability  $k$  defined by the material parameter `Permeability`, the viscosity  $\mu$  defined by the material parameter `Viscosity` and  $\Pi_0$  an additional mass production rate defined by the material parameter `Rate0`.

This problem description has to be completed by the user with BCs for the dof-field `Pressure` on  $\partial\Omega_{\text{porous}}$ . A Dirichlet BC sets the value of the pressure  $p$  and a Neumann BC sets the value of mass flow  $\rho \mathbf{v} \cdot \mathbf{n}$  with  $\mathbf{n}$  the normal to the boundary, see Table 5.1.

### Equation Transport< $\alpha$ >

This equation solves for the species mole fraction  $x_\alpha$  (Dof-field < $\alpha$ >) in the material domain  $\Omega_\alpha$  where < $\alpha$ > and  $\alpha$  are placeholders for the name and symbol of any species defined with the `Species` statement explained on p. 47. From the definition of  $x_\alpha$ , we obtain the following relation for the mass density of the mixture  $\rho$  defined by the material parameter `Density` and of its components  $\rho_\alpha$

$$\rho = \sum_{\alpha} \rho_{\alpha} \quad \text{with} \quad \rho_{\alpha} = \rho x_{\alpha} M_{\alpha} / M, \quad (5.9)$$

$M_{\alpha}$  the species molar masses defined by the material parameter `Mmol` and  $M \equiv \sum_{\alpha} M_{\alpha} x_{\alpha}$  the average molar mass. The mixing of species is described by the species mass conservation laws

$$\partial_t \rho_{\alpha} + \nabla \cdot (\rho_{\alpha} \mathbf{v}) = -\nabla \cdot \mathbf{j}_{\alpha} + \Pi_{\alpha}, \quad (5.10)$$

with  $\rho_{\alpha}$  the species mass density (5.9),  $\mathbf{v}$  the center of mass velocity to be defined e.g. by solving the Navier-Stokes eqs. (5.1)-(5.2),  $\mathbf{j}_{\alpha}$  the species diffusion current with respect to  $\mathbf{v}$  and  $\Pi_{\alpha}$  the species production rate defined by the material parameter `Rate`. The diffusion current  $\mathbf{j}_{\alpha}$  is given by

$$\mathbf{j}_{\alpha} = -\sum_{\beta} D_{\alpha\beta} \nabla x_{\beta}, \quad (5.11)$$

with  $D_{\alpha\beta}$  the diffusion coefficients defined by the material parameter `Diff`. The built-in function `StefanMaxwellDiff` is available to specify these coefficients after the Stefan-Maxwell model (5.73).

Together with the mass conservation (5.1), the species mass conservation (5.10) can be rewritten in the form

$$\partial_t \rho_{\alpha} - \frac{\rho_{\alpha}}{\rho} \partial_t \rho + \nabla \cdot \left( \frac{\rho_{\alpha}}{\rho} \cdot \rho \mathbf{v} \right) = -\nabla \cdot \mathbf{j}_{\alpha} + \Pi_{\alpha}. \quad (5.12)$$

Using the approximation  $\partial_t(\rho/M) = 0$  together with the relation (5.9) and the diffusion law (5.11), the above equation assumes the form

$$\rho \frac{M_{\alpha}}{M} \partial_t x_{\alpha} - \rho_{\alpha} \sum_{\beta} \frac{M_{\beta}}{M} \partial_t x_{\beta} + \nabla \cdot \left( \frac{x_{\alpha} M_{\alpha}}{M} \cdot \rho \mathbf{v} \right) = \nabla \cdot \sum_{\beta} D_{\alpha\beta} \nabla x_{\beta} + \Pi_{\alpha}. \quad (5.13)$$

The convective flux

$$\nabla \cdot \left( x_{\alpha} M_{\alpha} / M \right) \cdot \rho \mathbf{v} = \rho \mathbf{v} \frac{M_{\alpha}}{M} \left[ \nabla x_{\alpha} - x_{\alpha} \sum_{\beta} \frac{M_{\beta}}{M} \nabla x_{\beta} \right], \quad (5.14)$$

may be included in the definition of the rate  $\Pi_{\alpha}$ . However, for numerical stability reasons, it should be defined separately with the material parameter `ThermConv` in the form `ThermConv.Vel =  $\rho \mathbf{v}$ , ThermConv.< $\alpha$ >_D< $\beta$ >Grad =  $M_{\alpha}/M \delta_{\alpha\beta} - \sum_{\beta} M_{\beta}/M$  and ThermConv.< $\alpha$ >_D< $\beta$ > =  $\partial_{x_{\beta}} [\nabla(x_{\alpha} M_{\alpha} / M) \cdot \rho \mathbf{v}]$` . If the velocity  $\mathbf{v}$  is not identically zero, it is an error not to specify `TransConv` and if the convective flux is included within the rate, one has to declare the model `TransConvDefined`. The

stabilization of the convective flux defined by TransConv can be changed with the material parameter TransStab.

This problem description has to be completed by the user with BCs for the dof-field  $\langle \alpha \rangle$  on  $\partial\Omega_{\langle \alpha \rangle}$ . A Dirichlet BC sets the value of the mole fraction  $x_\alpha$  and a Neumann BC sets the value of the diffusion current  $\mathbf{j}_\alpha \cdot \mathbf{n}$  with  $\mathbf{n}$  the normal to the boundary, see Table 5.1.

The balance equations for the species are not independent from each other. Clearly the mole fractions must sum up to 1, i.e.  $\sum_\alpha x_\alpha = 1$ . In principle, it would be possible to eliminate a dof-field representing a mole fraction, however, there is not always a single species dominating everywhere so that solving for all the species mole fractions is often a necessity and the constraint  $\sum_\alpha x_\alpha = 1$  must be *implicitly* enforced. Here the user is responsible to set this condition at all Dirichlet boundaries i.e.  $\sum_\alpha x_\alpha|_{\partial G_D} = 1$  and to assure that the column's sum of  $D_{\alpha\beta}$  is the same for all columns otherwise the code behavior is undefined. The sum rule for the mole fractions *inside* the modeling domain  $G$  is then a simple consequence of the min-max-principle for elliptic partial differential equations. The proof relies on the fact that by assumption the sum  $\sum_\alpha D_{\alpha\beta} = C$  is independent of  $\beta$ . Summing over the terms in (5.13), we arrive at  $0 + \nabla 1 \cdot \rho \mathbf{v} = \nabla \cdot (\nabla C \sum_\alpha x_\alpha) + \sum_\alpha \Pi_\alpha$ . But since the net mass production in a chemical reaction is zero  $0 = \sum_\alpha \Pi_\alpha$ , we obtain  $0 = \nabla \cdot (\nabla C \sum_\alpha x_\alpha)$ . The solutions to this Poisson equation have their minimum and maximum values on the boundary, where  $\sum_\alpha x_\alpha$  equals 1, therefore  $\sum_\alpha x_\alpha = 1$  holds everywhere.

### Equation Convect $\langle \alpha \rangle$ [ 2 ]

This equation convects the field  $c_\alpha$  (Dof-field  $\langle \alpha \rangle$ ) in the material domain  $\Omega_{c_\alpha}$  along the streamline of the vector field  $\mathbf{V}$  using either first or 2nd order  $H^1(\Omega_{c_\alpha})$  FEs and where  $\langle \alpha \rangle$  and  $\alpha$  are placeholders for the name and symbol of any convected field defined with the ConvectDef statement explained on p. 48. We assume the vector field  $\mathbf{V}$  to be divergence free  $\nabla \cdot \mathbf{V} = 0$  so that the conservation law for  $c_\alpha$  without any physical diffusion is given by

$$\partial_t c_\alpha + \nabla c_\alpha \cdot \mathbf{V} = \delta_\alpha, \quad (5.15)$$

with the production rate  $\delta_\alpha$  defined by the material parameter Rate $\langle \alpha \rangle$  and the convecting vector field  $\mathbf{V}$  by the material parameter ConvectVelocity. The numerical implementation assumes  $\mathbf{V} \neq 0$ .

This problem description has to be completed by the user with BCs for the dof-field Convect $\langle \alpha \rangle$  only on inflow boundaries  $\{\mathbf{x} \in \partial\Omega_{c_\alpha} : \mathbf{V} \cdot \mathbf{n} < 0\}$  with  $\mathbf{n}$  the normal to the boundary. A Dirichlet BC sets the value of  $c_\alpha$  and a Neumann BC sets the value of the flux  $c_\alpha \mathbf{V} \cdot \mathbf{n}$ , see Table 5.1.

### Equation ThermalEnergy [ 2 ]

These equations solve the first law of thermo-dynamic and compute the temperature  $T$  (Dof-field Temp) in the thermal conducting domain  $\Omega_{\text{therm}}$  using either first or 2nd

order  $H^1(\Omega_{\text{therm}})$  FEs. The governing equation is obtained starting from the conservation law for the total enthalpy  $\rho h = \sum_{\alpha} \rho_{\alpha} h_{\alpha}$

$$\partial_t \sum_{\alpha} \rho_{\alpha} h_{\alpha} + \nabla \cdot \sum_{\alpha} h_{\alpha} (\rho_{\alpha} \mathbf{v} + \mathbf{j}_{\alpha}) = D_t p - (\boldsymbol{\tau} \cdot \nabla) \mathbf{v} - \nabla \cdot \mathbf{F} + q_0, \quad (5.16)$$

with  $\rho_{\alpha}$  the species density (5.9),  $h_{\alpha}$  the species enthalpy per unit mass defined the material parameter `Enthalpy`,  $D_t p = \partial_t p + \mathbf{v} \cdot \nabla p$  the total time derivative of the pressure,  $\mathbf{v}$  the fluid velocity defined e.g by solving the Navier-Stokes eqs. (5.1)-(5.2),  $\mathbf{j}_{\alpha}$  the species diffusion currents (5.11) with respect to  $\mathbf{v}$ ,  $\mathbf{F}$  the conductive heat flux and  $q_0$  the heat source. The right hand side of eq. (5.16) describes the enthalpy change by expansion or compression of the moving fluid, irreversible production of heat due to mechanical friction, conductive heat transfer, convection by interdiffusion of species and a heat source. The contribution of mechanical friction  $(\boldsymbol{\tau} \cdot \nabla) \mathbf{v}$  to heat production is for most applications negligible as well as the the reversible exchange  $D_t p$  due to expansion or compression which therefore are neglected. The conductive heat flux is given by Fourier's law of heat transport

$$\mathbf{F} = -\boldsymbol{\kappa} \cdot \nabla T, \quad (5.17)$$

with  $\boldsymbol{\kappa}$  the thermal conductivity defined by the material parameters `KappaIso` or `KappaAni` whenever the tensor is isotropic or anisotropic. With the species mass conservation (5.10), eq. (5.16) can now be rewritten in the form

$$[(\rho c_P)_0 + \sum_{\alpha} \rho_{\alpha} \partial_T h_{\alpha}] \partial_t T + \sum_{\alpha} \nabla h_{\alpha} (\rho_{\alpha} \mathbf{v} + \mathbf{j}_{\alpha}) - \nabla \cdot (\boldsymbol{\kappa} \cdot \nabla T) = q_0 - \sum_{\alpha} h_{\alpha} \Pi_{\alpha} = q, \quad (5.18)$$

with an amended heat source  $q$  defined by the material parameter `Heat` where the net enthalpy production  $\sum_{\alpha} h_{\alpha} \Pi_{\alpha}$  of a chemical reaction need to be subtracted and  $(\rho c_P)_0$  is an additional residual or solid matter specific heat defined by the material parameter `CpDensity0`. The convection term  $\sum_{\alpha} \nabla h_{\alpha} \cdot (\rho_{\alpha} \mathbf{v} + \mathbf{j}_{\alpha})$  may be included in the definition of the heat source. However, for numerical stability reasons, it should be defined separately with the material parameter `ThermConv` in the form  $\text{ThermConv} = \sum_{\alpha} \partial_T h_{\alpha} (\rho_{\alpha} \mathbf{v} + \mathbf{j}_{\alpha})$ . If the enthalpy has other dependencies than the temperature, the additional variations  $\sum_{\alpha} \partial_{<\bullet>} h_{\alpha} (\rho_{\alpha} \mathbf{v} + \mathbf{j}_{\alpha}) \cdot \nabla <\bullet>$  need to be included in the heat source. If the velocity  $\mathbf{v}$  or the species diffusion currents  $\mathbf{j}_{\alpha}$  are not identically zero, it is an error not to specify `ThermConv` and if the convective flux is included within the heat source, one has to declare the model `ThermConvDefined`. The stabilization of the convective flux defined by `ThermConv` can be changed with the material parameter `ThermStab`. For a stationary solution, the model parameters  $(\rho c_P)_0$ ,  $h_{\alpha}$  and  $\rho$  are not directly accessed.

This problem description has to be completed by the user with BCs for the dof-field `Temp` on  $\partial\Omega_{\text{therm}}$ . A Dirichlet BC sets the value of the temperature  $T$  and a Neumann BC sets the value of the conductive heat flux  $\mathbf{F} \cdot \mathbf{n}$  with  $\mathbf{n}$  the normal to the boundary, see Table 5.1.

## Equation `OhmicCurrent` [ 2 ]

These equations solve the Maxwell's current law for conductors and compute the electric potential  $\Psi$  (Dof-field `Psi`) in the conducting domain  $\Omega_{\text{ohm}}$  using either first or 2nd

order  $H^1(\Omega_{\text{ohm}})$  FEs. In this domain, the quasi stationary electric behavior is governed by the conservation of the current density  $\mathbf{J}$

$$\nabla \cdot \mathbf{J} = q_0 G, \quad (5.19)$$

with  $q_0$  the elementary charge and  $G$  the generation rate specified by the material parameter `Gen`. The current  $\mathbf{J}$  is related to the electric field  $\mathbf{E} = -\nabla \Psi$  by the Ohm's law

$$\mathbf{J} = \boldsymbol{\sigma} \cdot \mathbf{E} + \mathbf{J}_{00} = -\boldsymbol{\sigma} \cdot \nabla \Psi + \mathbf{J}_{00}, \quad (5.20)$$

with  $\boldsymbol{\sigma}$  the electric conductivity defined by the material parameters `SigmaIso`, `SigmaAni` or `SigmaUns` depending on whether the tensor is isotropic, anisotropic symmetric or anisotropic unsymmetric. In addition,  $\mathbf{J}_{00}$  is the external current which may be specified by the material parameter `Current00`.

This problem description has to be completed by the user with BCs for the dof-field `Psi` on  $\partial\Omega_{\text{ohm}}$ . A Dirichlet BC sets the value of the electric potential  $\Psi$  and a Neumann BC sets the value of the current  $\mathbf{J} \cdot \mathbf{n}$  with  $\mathbf{n}$  the normal to the boundary, see Table 5.1.

### Equation ElectroStatic[2]

These equations solve the Maxwell's electrostatic law and compute the electrostatic potential  $\Phi$  (Dof-field `Phi`) in the non-conducting domain  $\Omega_{\text{elstat}}$  using either first or 2nd order  $H^1(\Omega_{\text{elstat}})$  FEs. In this domain, the electrostatic behavior is governed by Maxwell's equation for the dielectric displacement field  $\mathbf{D}$  given by

$$\nabla \cdot \mathbf{D} = q_0 \varrho, \quad (5.21)$$

with  $\varrho$  a space charge concentration defined by the material parameter `Charge` and  $q_0$  the elementary charge. The dielectric displacement  $\mathbf{D}$  is related to the electrostatic potential  $\Phi$  by the linear material law

$$\mathbf{D} = \boldsymbol{\epsilon} \cdot \mathbf{E} = -\boldsymbol{\epsilon} \cdot \nabla \Phi, \quad (5.22)$$

with  $\boldsymbol{\epsilon} = \epsilon_0 \boldsymbol{\epsilon}_{\text{rel}}$  the dielectric permittivity,  $\epsilon_0 = 8.85418 \cdot 10^{-12} \text{ C}/(\text{Vm})$  the vacuum dielectric constant and  $\boldsymbol{\epsilon}_{\text{rel}}$  the relative dielectric permittivity defined by the material parameters `EpsIso` or `EpsAni` whenever the tensor is isotropic or anisotropic.

This problem description has to be completed by the user with BCs for the dof-field `Phi` on  $\partial\Omega_{\text{elstat}}$ . A Dirichlet BC sets the value of the electrostatic potential  $\Phi$  and a Neumann BC sets the value of the surface charge  $-\mathbf{D} \cdot \mathbf{n}$  with  $\mathbf{n}$  the normal to the boundary, see Table 5.1. The global parameter `PhiEqScaling` is available to additionally scale the eq. (5.21), a useful tool used for symmetrization in coupled problems.

### Equation MagnetoStatic[2]

These equations solve the Maxwell's magnetostatic law and compute the reduced or total magnetic potential  $\Theta$  (Dof-field `MagnPot`) in the domain  $\Omega_{\text{magn}}$  using either first



or 2nd order  $H^1(\Omega_{\text{magn}})$  FEs. In this domain, the magnetostatic behavior is governed by Maxwell's equations

$$\nabla \cdot \mathbf{B} = 0, \quad \nabla \times \mathbf{H} = \mathbf{J}_0, \quad (5.23)$$

for the induction field  $\mathbf{B}$ , the magnetic field  $\mathbf{H}$  and a given external current  $\mathbf{J}_0$  with  $\nabla \cdot \mathbf{J}_0 = 0$ . By splitting the magnetic field in two parts  $\mathbf{H} = \mathbf{H}_0 + \mathbf{H}_1$  and defining  $\mathbf{H}_0$  through the relation  $\mathbf{J}_0 = \nabla \times \mathbf{H}_0$ , we obtain  $\nabla \times \mathbf{H}_1 = 0$ , so that for  $\mathbf{H}_1$  a magnetic scalar potential  $\Theta$  can be defined with  $\mathbf{H}_1 = -\nabla \Theta$  and  $\mathbf{H} = \mathbf{H}_0 - \nabla \Theta$ . Using the material law  $\mathbf{B} = \mu \mathbf{H}$  with  $\mu$  the permeability and the solenoidal property  $\nabla \cdot \mathbf{B} = 0$ , one obtains the Poisson equation

$$\nabla \cdot (\mu \nabla \Theta - \mu \mathbf{H}_0) = 0, \quad (5.24)$$

for the reduced scalar potential  $\Theta$  represented by the dof-field `MagnPot`. The relative permeability  $\mu_{\text{rel}} = \mu/\mu_0$  is defined by the material parameter `Mue` with  $\mu_0 = 4\pi \times 10^{-7} \text{ Vs/(Am)}$  and the external field  $\mathbf{H}_0$  by the material parameter `Hfield0`. For a non-constant permeability  $\mu_{\text{rel}} = \mu_{\text{rel}}(|\mathbf{H}|)$ , the logarithmic derivative  $|\mathbf{H}|^{-1} \partial \mu_{\text{rel}} / \partial |\mathbf{H}|$  should be specified with the material parameter `Mue.DlogH`. The  $\mathbf{B}$ -field is not available to specify  $\mu_{\text{rel}}$ .

This problem description has to be completed with BCs for the dof-field `MagnPot` on  $\partial\Omega_{\text{magn}}$ . A Dirichlet BC sets the value of the magnetic potential  $\Theta$  and a Neumann BC sets the value of the induction field  $\mathbf{B} \cdot \mathbf{n}$  with  $\mathbf{n}$  the normal to the boundary, see Table 5.1. The global parameter `MagnPotEqScaling` is available to additionally scale the eq. (5.24), a useful tool used for symmetrization in coupled problems.

The field  $\mathbf{H}_0$  is given by the Biot-Savart's law and direct integration of  $\mathbf{J}_0$  as

$$\mathbf{H}_0(\mathbf{x}) = \frac{1}{4\pi} \int_{\mathbb{R}^3} \frac{\mathbf{J}_0(\mathbf{y}) \times (\mathbf{x} - \mathbf{y})}{|\mathbf{x} - \mathbf{y}|^3} dV_y. \quad (5.25)$$

For a 2D domain, we must assume  $\mathbf{J}_0(\mathbf{y}) = J_0(\mathbf{y})\mathbf{e}_z$  and  $z$ -axis invariance of the current. Integrating along the  $z$ -axis yields the somewhat modified Biot-Savart's law

$$\mathbf{H}_{0,2D}(\mathbf{x}) = \frac{1}{2\pi} \int_{\mathbb{R}^2} \frac{\mathbf{J}_0(\mathbf{y}) \times (\mathbf{x} - \mathbf{y})}{|\mathbf{x} - \mathbf{y}|^2} dV_y. \quad (5.26)$$

For the field  $\mathbf{H}_0$ , we have the property  $\nabla \times \mathbf{H}_0 = \mathbf{J}_0$  but only if  $\nabla \cdot \mathbf{J}_0 = 0$  holds everywhere. This is always true for the 2D case, however, for the 3D case, this may not be satisfied for a general user defined current density  $\mathbf{J}_0$ .

The computed field  $\mathbf{H}_1 = -\nabla \Theta$  is interpreted as the correction to the magnetostatic solution in free space  $\mathbf{H}_0$  when considering BCs and material laws. This formulation is very attractive since it is simple and solves for the reduced scalar potential  $\Theta$ , a well defined function if the domain  $\Omega_{\text{magn}}$  is simply connected. For the simple material law  $\mu = \text{const}$ , we have  $0 = \mu \nabla \cdot \mathbf{H}_0 = \nabla \cdot \mu \mathbf{H}_0$  so that the term  $\mathbf{H}_0$  in (5.24) may be eventually replaced by Neumann BCs. For magnetic materials of non-constant permeability  $\mu$ , this procedure does not apply and for high permeability  $\mu \gg 1$ , the fields  $\nabla \Theta$  and  $\mathbf{H}_0$  are of the same magnitude but of opposite direction so that numerical cancellation in (5.24) may completely destroy the solution. In order to compute numerical stable solutions, we must assume zero external currents in these critical regions of high permittivity denoted by  $\Omega_\mu$ . From the property  $\nabla \times \mathbf{H} = 0$ , a total scalar potential



$\Theta_{\text{tot}}$  can be defined with  $\mathbf{H} = -\nabla\Theta_{\text{tot}}$  and computed by solving the Poisson equation  $\nabla \cdot \mu \nabla \Theta_{\text{tot}} = 0$ , thus avoiding numerical cancellation in the subdomain  $\Omega_\mu$ . Now we need to connect both formulations for  $\Theta$  in  $\Omega_{\text{magn}} \setminus \Omega_\mu$  and for  $\Theta_{\text{tot}}$  in  $\Omega_\mu$ . The governing equations (5.23) imply the normal component of  $\mathbf{B}$  and the tangential component of  $\mathbf{H}$  to be continuous at the interface  $\partial\Omega_\mu$ . The first condition is automatically implied by the finite element method while the second must be imposed separately as  $(\nabla\Theta - \mathbf{H}_0 - \nabla\Theta_{\text{tot}}) \cdot \mathbf{t} = 0$  with  $\mathbf{t}$  any tangent vector to the surface  $\partial\Omega_\mu$ . This condition has a simpler formulation if we assume  $\mathbf{J}_0 = 0$  in the subdomain  $\Omega_\mu$  so that the field  $\mathbf{H}_0$  is a gradient field defined by the Biot Savart's potential  $\Theta_0$  as  $\mathbf{H}_0 = -\nabla\Theta_0$ . On the boundary  $\partial\Omega_\mu$  we thus have the relation

$$\Theta(\mathbf{x}) = \Theta_{\text{tot}}(\mathbf{x}) - \Theta_0(\mathbf{x}). \quad (5.27)$$

Both formulations for the reduced and total scalar potentials can be unified in a single one by considering the single potential  $\Theta$  as a double valued function at the interface  $\partial\Omega_\mu$  with a discontinuity of  $\Theta_0$  and by considering  $\mathbf{H}_0 = 0$  in  $\Omega_\mu$ . The Biot Savart's potential is well defined on  $\bar{\Omega}_\mu$  and given by a path integral

$$\Theta_0(\mathbf{x}) = \int_{\gamma(\mathbf{x} \rightsquigarrow \mathbf{x}_0)} \mathbf{H}_0 \cdot d\gamma, \quad (5.28)$$

with  $\mathbf{x}_0, \mathbf{x} \in \bar{\Omega}_\mu$  and  $\gamma(\mathbf{x} \rightsquigarrow \mathbf{x}_0)$  any path on  $\bar{\Omega}_\mu$  joining  $\mathbf{x}$  to the fixed point  $\mathbf{x}_0$ . For numerical purposes, we choose  $\mathbf{x}_0$  at infinity and the path  $\gamma = \mathbf{x} + t\mathbf{n}$  with  $t \in [0, \infty]$  resulting in the integral

$$\Theta_0(\mathbf{x}) = \frac{1}{4\pi} \int_{\mathbb{R}^3} dV_y \mathbf{J}_0(\mathbf{y}) \int_0^\infty \frac{(\mathbf{x} - \mathbf{y})}{|\mathbf{x} - \mathbf{y} + t\mathbf{n}|^3} \times \mathbf{n} dt = \int_{\mathbb{R}^3} \mathbf{J}_0(\mathbf{y}) \cdot \mathbf{w}(\mathbf{x}, \mathbf{y}) dV_y, \quad (5.29)$$

with

$$\mathbf{w}(\mathbf{x}, \mathbf{y}) = \frac{(\mathbf{x} - \mathbf{y}) \times \mathbf{n}}{|\mathbf{x} - \mathbf{y}|(\mathbf{n} \cdot (\mathbf{x} - \mathbf{y}) + |\mathbf{x} - \mathbf{y}|)}. \quad (5.30)$$

The field  $\mathbf{w}(\mathbf{x}, \mathbf{y})$  is not a gradient field and  $\Theta_0(\mathbf{x})$  is the correct potential field only if  $\nabla \cdot \mathbf{J}_0 = 0$  holds everywhere on  $\mathbb{R}^3$ . From our assumption  $\mathbf{J}_0 = 0$  on  $\Omega_\mu$ , we always have  $\mathbf{x} \neq \mathbf{y}$  so that the function (5.30) can only be singular if  $\mathbf{n}|\mathbf{x} - \mathbf{y}| = \mathbf{y} - \mathbf{x}$  i.e. the path of integration crosses the domain  $\text{supp}(\mathbf{J}_0)$ . For a 2D domain, we have  $\mathbf{w}(\mathbf{x}, \mathbf{y}) = (0, 0, \text{atan2}(R(\mathbf{n})(\mathbf{x} - \mathbf{y})))$  and  $R(\mathbf{n})$  a rotation matrix so that the function  $\text{atan2}$  is discontinuous along the  $-\mathbf{n}$  direction. The direct computation of  $\Theta_0$  by (5.29) may be unstable for several reasons, in this case the potential  $\Theta_0$  should be computed in a preprocessing step by solving the Laplace equation  $\nabla \cdot \nabla \Theta_0 = 0$  on  $\Omega_\mu$  with the Neumann BC  $\mu_0 \mathbf{H}_0 \cdot \mathbf{n}$  on  $\partial\Omega_\mu$  and  $\mathbf{n}$  the outward normal to  $\partial\Omega_\mu$ .

### Equation EddyCurrent [Nodal] [2] in 3D

These equations solve the governing equations for eddy current analysis in conductor domains  $\Omega_{\text{eddy}}$  and compute the vector potential  $\mathbf{A}$  (Dof-field `Afield.<T1>`) and the time-integral of the scalar potential  $\bar{\Psi} = \int \Psi dt$  (Dof-field `Vint`) using either  $H(\text{curl}) \times H^1$  or  $(H^1)^3 \times H^1$  FEs of first or 2nd order. Starting with the Maxwell's equations, for an eddy current analysis we assume a negligible displacement current  $\partial_t \mathbf{D}$ , the Ohm's law  $\mathbf{J} = \sigma \mathbf{E}$  and material law  $\mathbf{B} = \mu \mathbf{H}$  so that the equations left to solve are given by

$$\nabla \cdot \mathbf{B} = 0, \quad \nabla \times \mathbf{E} + \partial_t \mathbf{B} = 0, \quad \nabla \times \mathbf{H} = \mathbf{J} = \sigma \mathbf{E} + \mathbf{J}_0 + \nabla \times \mathbf{H}_0, \quad (5.31)$$

with  $\mathbf{J}_0$  an external driving current satisfying  $\nabla \cdot \mathbf{J}_0 = 0$  and  $\mathbf{H}_0$  an external  $\mathbf{H}$ -field possibly representing a magnetization or the Biot-Savart law (5.25) for a given  $\mathbf{J}_0 = \nabla \times \mathbf{H}_0$  in order for the discretization to be independent from  $\text{supp}(\mathbf{J}_0)$ . The values  $\mu$ ,  $\sigma$ ,  $\mathbf{J}_0$  and  $\mathbf{H}_0$  are defined by the material parameter `Mue`, `SigmaIso`, `Current0` and `Hfield0`. For a non-constant permeability  $\mu_{\text{rel}} = \mu_{\text{rel}}(|\mathbf{B}|)$ , the logarithmic derivative  $|\mathbf{B}|^{-1} \partial \mu_{\text{rel}} / \partial |\mathbf{B}|$  should be specified with the material parameter `Mue.DlogB`. The  $\mathbf{H}$ -field is not available to specify  $\mu_{\text{rel}}$ .

Using the solenoidal property  $\nabla \cdot \mathbf{B} = 0$ , we can define a vector potential  $\mathbf{A}$  with the property  $\mathbf{B} = \nabla \times \mathbf{A}$  and because of the relation  $\nabla \times (\mathbf{E} + \partial_t \mathbf{A}) = 0$ , we can define a scalar potential  $\Psi$  with

$$-\nabla \Psi = \mathbf{E} + \partial_t \mathbf{A}, \quad (5.32)$$

and rewrite (5.31) in the form

$$\nabla \times \mu^{-1} \nabla \times \mathbf{A} + \sigma \partial_t \mathbf{A} + \sigma \nabla \Psi - \mathbf{J}_0 - \nabla \times \mathbf{H}_0 = 0. \quad (5.33)$$

Although the condition  $\nabla \cdot \mathbf{J} = 0$  follows from (5.33), we use it explicitly in order to arrive at a solvable system and for symmetry reasons we use the time-integral of the potential  $\bar{\Psi} = \int \Psi dt$  thus obtaining

$$\begin{cases} \nabla \times \mu^{-1} \nabla \times \mathbf{A} + \partial_t (\sigma \mathbf{A} + \sigma \nabla \bar{\Psi}) - \mathbf{J}_0 - \nabla \times \mathbf{H}_0 = 0, \\ \partial_t \nabla \cdot (\sigma \mathbf{A} + \sigma \nabla \bar{\Psi}) = 0. \end{cases} \quad (5.34)$$

Since the physical solution is unchanged by adding any  $\bar{\Psi}_0$  to  $\bar{\Psi}$  and  $-\nabla \bar{\Psi}_0$  to  $\mathbf{A}$ , this formulation is ungauged and in this form it is discretized using  $H(\text{curl}; \Omega_{\text{eddy}})$  FEs for  $\mathbf{A}$  and  $H^1(\Omega_{\text{eddy}})$  FEs for  $\bar{\Psi}$ . The fields  $\mathbf{A}$  and  $\bar{\Psi}$  are not uniquely determined and so iterative linear solvers are required for solving the singular system of equations.

We may as well enforce the Coulomb gauge  $\nabla \cdot \mathbf{A} = 0$  and so uniqueness of  $\mathbf{A}$  and  $\bar{\Psi}$  by solving the equations

$$\begin{cases} \nabla \times \mu^{-1} \nabla \times \mathbf{A} - \mu_*^{-1} \nabla (\nabla \cdot \mathbf{A}) + \partial_t (\sigma \mathbf{A} + \sigma \nabla \bar{\Psi}) - \mathbf{J}_0 - \nabla \times \mathbf{H}_0 = 0, \\ \partial_t \nabla \cdot (\sigma \mathbf{A} + \sigma \nabla \bar{\Psi}) = 0, \end{cases} \quad (5.35)$$

with the constant  $\mu_*$  a suitable average of  $\mu$ . By taking the divergence of (5.35), we obtain  $\nabla \cdot \nabla (\nabla \cdot \mathbf{A}) = 0$  so that  $\nabla \cdot \mathbf{A} = 0$  holds on  $\Omega_{\text{eddy}}$  by a correct choice of BCs. This Coulomb gauge formulation is discretized using  $H^1(\Omega_{\text{eddy}})$  FEs for the components of  $\mathbf{A}$  and  $\bar{\Psi}$  and is used whenever the `<Nodal>` name's option is given.

This problem description has to be completed by the user with BCs for the dof-field `Afield` and `Vint` on  $\partial \Omega_{\text{eddy}}$ . For the vector potential  $\mathbf{A}$ , a Dirichlet BC sets the value of the vector potential  $\mathbf{A}$  and a Neumann BC sets the tangent value  $(\mathbf{H} - \mathbf{H}_0) \times \mathbf{n}$  with  $\mathbf{n}$  the normal to the boundary, see Table 5.1.

### Equation `EddyFree[Nodal][2]` in 3D

These equations solve the same models of (5.34)-(5.35) by considering a zero potential  $\Psi = 0$ . Let assume  $\sigma = \text{const}$ , then the condition  $\nabla \cdot \mathbf{J} = 0$  gives  $\nabla \cdot \mathbf{E} = 0$  and by considering the Coulomb gauge  $\nabla \cdot \mathbf{A} = 0$ , from (5.32) we have  $\nabla \cdot \nabla \Psi = 0$ . If on the

boundary  $\partial\Omega_{\text{eddy}}$  we can assume  $\Psi = 0$ , then we have  $\Psi = 0$  on  $\Omega_{\text{eddy}}$  and we are left to solve

$$\nabla \times \mu^{-1} \nabla \times \mathbf{A} + \partial_t \sigma \mathbf{A} - \mathbf{J}_0 - \nabla \times \mathbf{H}_0 = 0, \quad (5.36)$$

or

$$\nabla \times \mu^{-1} \nabla \times \mathbf{A} - \mu_*^{-1} \nabla (\nabla \cdot \mathbf{A}) + \partial_t \sigma \mathbf{A} - \mathbf{J}_0 - \nabla \times \mathbf{H}_0 = 0, \quad (5.37)$$

whenever the `<Nodal>` name's option is given. With this formulation, we can save on the computation of the potential  $\Psi$  but we are less flexible in the setting of practical BCs, it is however well suited for the free space with  $\sigma = 0$ . When using (5.36) together with a  $H(\text{curl}; \Omega_{\text{eddy}})$  FE and  $\sigma = 0$ , one has to consider a large null-space of the system matrix arising from the non-trivial kernel of the  $\nabla \times H(\text{curl})$  operator and again iterative linear solvers are required for solving the equations. Also one should not use  $\mathbf{J}_0$  unless it lies in the space  $\nabla \times H(\text{curl})$  in order for the system right-hand side to lie in the image space of the system matrix. Differently, the case  $\sigma \neq 0$  always yields regular matrices. It is to be noted, that in order to further reduce the computational work, for the free space case with  $\sigma = 0$  and  $\mathbf{J}_0 = 0$ , it is possible to replace the computation of  $\mathbf{A}$  by a scalar formulation computing the magnetic potential  $\Theta$  of (5.24). The drawback is that one has to define interface conditions between the fields  $\Theta$  and  $\mathbf{A}$ ,  $\bar{\Psi}$  and since the domain is in general not simple connected also jump conditions for  $\Theta$ .

### Equation EddyCurrent[2] in 2D

In a 2D domain, the current flow is assumed normal to the domain  $\mathbf{J} = (0, 0, J_z)$  and the  $\mathbf{H}$ -field in-plane. We therefore have  $\mathbf{E} = (0, 0, E_z)$ ,  $\mathbf{A} = (0, 0, A_z)$ ,  $\nabla_2 \Psi = 0$  and we can assume a zero potential  $\Psi = 0$ . Together with the relation  $\nabla \times \mu^{-1} \nabla \times \mathbf{A} = -(\nabla \mu^{-1} \nabla) \mathbf{A} + \nabla \mu^{-1} \nabla \cdot \mathbf{A}$ , we are left to solve the single scalar equation

$$-\nabla \mu^{-1} \nabla A_z + \sigma \partial_t A_z - J_{0z} - (\nabla \times \mathbf{H}_0)_z = 0. \quad (5.38)$$

For an axis-symmetric eddy current analysis using the global model `AxisSymmetric`, see p. 135, the equation (5.38) must be rewritten into cylindrical coordinates. However, since the 2D differential operator  $\nabla \cdot \mu^{-1} \nabla(\bullet)$  stems from the 3D operator  $\nabla \times \mu^{-1} \nabla \times (\bullet)$ , a total different situation arises from the one for the Laplace operator which does not require any particular user's intervention. When the axisymmetric model is enabled, instead of computing  $A_z$ , we actually compute the field  $\rho A_z$  and this difference is not yet taken into account when defining Dirichlet BCs and initial values for the dof-field `Afield` so that values must be specified for  $\rho A_z$  instead of  $A_z$ . Further, Neumann BCs for `Afield` should not be used because they are not yet correctly implemented.

### Equation Eddy<...>Harmonic[2] MagnetoStaticHarmonic[2]

All the previous `Eddy<...>[2]` and `MagnetoStatic[2]` equations for eddy current and magnetostatic computations have a counterpart `Eddy<...>Harmonic[2]`

and `MagnetoStaticHarmonic[2]` for the case of a time dependent harmonic analysis. We work with complex values and a time dependency of  $(\bullet)(t) = (\bullet)e^{i\omega t}$  for  $\mathbf{A}$ ,  $\Psi$ ,  $\mathbf{J}_0$ ,  $\mathbf{H}_0$ ,  $\Theta$  with  $\omega$  a fixed angular frequency defined by the global parameter `Frequency`. The equations to solve are directly obtained by substituting in (5.24), (5.34), (5.35), (5.36), (5.37) and (5.38), the time derivative  $\partial_t(\bullet)$  with  $i\omega(\bullet)$ . The real part of a complex value is accessed as before for a non-harmonic analysis, whereas the imaginary part is available by using the prefix `i`, e.g. `Current`, `iCurrent` represent the real and imaginary parts of the complex current  $\mathbf{J}$ . For a harmonic analysis, we have to assume linearity and thus the material parameters  $\sigma$ ,  $\mu$  can only be functions of the coordinates  $\mathbf{x}$  and the frequency  $\omega$ . The harmonic analysis doubles the number of unknown variables, does not require any time-integration algorithm, but in general requires a frequency sweep.

The global parameter `iAFIELDScaling` is available to additionally scale the imaginary part of eqs. (5.36), (5.37), (5.38), a useful tool used for symmetrization in coupled problems.

### Equation Elasticity[2,EAS]

These equations solve the elasticity laws of structure mechanics and compute the displacement  $\mathbf{u} = (u_x, u_y, u_z)^T$  (Dof-field `Disp.<T1>`) in the mechanical domain  $\Omega_{\text{mech}}$  using either first or 2nd order  $(H^1(\Omega_{\text{mech}}))^3$  displacement FEs or first order *enhanced assumed strain* FEs. The latter is a mixed FE with hidden internal variables, here the global model `EasStore` is available to speed-up the assembling process, see p. 136. When external body forces  $\mathbf{f}^\varphi$  are applied, a body originally located at  $\Omega_{\text{mech}}$  deforms and occupies the new region  $\Omega_{\text{mech}}^\varphi = \varphi(\Omega_{\text{mech}})$  with the deformation  $\varphi(\mathbf{X}) = \mathbf{x}(\mathbf{X}) = \mathbf{X} + \mathbf{u}(\mathbf{X})$  here expressed as function of the reference coordinates  $\mathbf{X} \in \Omega_{\text{mech}}$ . By applying the conservation of force and moment over the deformed domain  $\Omega_{\text{mech}}^\varphi$ , we obtain the following governing equations

$$\rho_0^\varphi D_t \mathbf{v} = \nabla_\varphi \cdot \mathbf{s} + \mathbf{f}^\varphi, \quad (5.39)$$

with  $\mathbf{s}$  the Cauchy stress tensor,  $\mathbf{v} = \partial_t \mathbf{u}$  the velocity,  $D_t(\bullet) = \partial_t(\bullet) + \nabla(\bullet) \cdot \mathbf{v}$  the material time derivative and  $\rho_0^\varphi$  the solid matter density, see [10]. In this formulation, the governing equations are expressed in term of the unknown deformation  $\varphi$  and partial derivatives  $\partial_\varphi(\bullet)$ . Therefore using the Piola identity  $\partial_i((\nabla\varphi)^{-1} \det \nabla\varphi)_{ij} = 0$ , they are conveniently rewritten for the reference and underformed domain  $\Omega_{\text{mech}}$  as

$$\rho_0 D_t \mathbf{v} = \nabla \cdot (\nabla\varphi \mathbf{S}) + \mathbf{f}, \quad (5.40)$$

with  $\mathbf{f} = \mathbf{f}^\varphi \det \nabla\varphi$  the body force per unit of volume,  $\rho_0 = \rho_0^\varphi \det \nabla\varphi$  the density and the second Piola-Kirchhoff stress tensor

$$\mathbf{S} = (\det \nabla\varphi) (\nabla\varphi)^{-1} \cdot \mathbf{s} \cdot (\nabla\varphi)^{-T}. \quad (5.41)$$

Although the second Piola-Kirchhoff stress tensor  $\mathbf{S}$  is symmetric, it bears no physical meaning compared to the true physical Cauchy stress  $\mathbf{s}$ . To relate the tensor  $\mathbf{S}$  with the deformation  $\varphi$ , we use the Green-Lagrange strain tensor

$$\mathbf{E} = \frac{(\nabla\varphi)^T \cdot (\nabla\varphi) - \mathbf{Id}}{2} = \frac{\nabla\mathbf{u} + (\nabla\mathbf{u})^T + (\nabla\mathbf{u})^T \cdot (\nabla\mathbf{u})}{2}, \quad (5.42)$$

and the material law  $\mathbf{S} = \mathbf{S}(\mathbf{E})$  together with the derivative  $\mathbf{C} = \partial \mathbf{S} / \partial \mathbf{E}$  are defined by the material parameters `StressGen`, `StressSym` or `StressOrtho` whenever the elasticity tensor  $\mathbf{C}$  is generic with just minor symmetries  $C_{ijkl} = C_{jikl}$  and  $C_{ijkl} = C_{ijlk}$ , has in addition major symmetries  $C_{ijkl} = C_{klij}$  or has minor and major symmetries with zero values for  $C_{iixy} = C_{iiyz} = C_{iixz} = C_{xyyz} = C_{xyxz} = C_{yzxz} = 0$ . The body force  $\mathbf{f}$  is defined by the material parameter `Force`. For small deformations  $\mathbf{u} = \boldsymbol{\varphi} - \mathbf{X}$ , the non-linear equations (5.40) yield results close to the geometrically linearized equations

$$\rho_0 \partial_t^2 \mathbf{u} = \nabla \cdot \mathbf{s}(\boldsymbol{\varepsilon}) + \mathbf{f}, \quad (5.43)$$

with the stress  $\mathbf{s} = \mathbf{s}(\boldsymbol{\varepsilon})$  now being a function of the linearized or engineering strain tensor

$$\boldsymbol{\varepsilon} = \frac{(\nabla \mathbf{u})^T + \nabla \mathbf{u}}{2}, \quad (5.44)$$

and representing all form of stresses which are in this formulation all equivalent. For a more complete analysis of elasticity theory as well as the computation of solutions by finite element methods, see for example [3, 8, 10, 17]. The same material parameters `StressGen`, `StressSym`, `StressOrtho` and `Force` as for the non-linear formulation are used to specify the material law and body force.

This problem description for the non-linear equations (5.40) or the linearized ones (5.43) has to be completed with boundary conditions for the dof-field `Disp.<T1>` on the undeformed boundary  $\partial \Omega_{\text{mech}}$ . A Dirichlet BC sets the value of the displacement  $\mathbf{u}$  and a Neumann BC sets the value of the surface traction  $\nabla \varphi \mathbf{S} \cdot \mathbf{n}$  in the reference configuration  $\Omega_{\text{mech}}$  with  $\mathbf{n}$  the normal to the undeformed boundary  $\partial \Omega_{\text{mech}}$ , see Table 5.1.

### Equation `ElasticShell[Single][I,J,K]`

These equations solve the same governing equations of the previous `Elasticity` model but compute the displacement  $\mathbf{u}$  (Dof-field `Disp.<T1>`) using solid-shell FEs. These elements do not make use of shell theories, work with 3D material laws but are inherently anisotropic since in order to fix the locking behavior of low order FEs, they assume a transverse shallow direction where the mechanical structure is assumed to be thin. Therefore, the three different versions are for shells where the shallowness is in the block `I`-, `J`- or `K`-direction. At the present time, the solid-shell elements are the one given in [11] and the degree-of-freedom are the same as for first order  $(H^1(\Omega_{\text{mech}}))^3$  FEs. Solid-shells are mixed FEs with hidden internal variables, here the global model `ShellStore` is available to speed-up the assembling process, see p. 136.

Without the option `Single`, the complexity of using solid-shell elements is the same as for using first order  $(H^1(\Omega_{\text{mech}}))^3$  FEs. If a shell is composed of several layers e.g. of different materials, each element in the transverse direction is a single solid-shell element. However, solid-shell FEs are devised to work with just a single element in the transverse shallow direction and this is the case when using the `Single` option. Here the displacement degree-of-freedom are just defined on the top and bottom of the shell surface and the stack of transverse elements acts as a single element. With this option enabled, adding elements in the transverse direction just results in a more accurate numerical integration but the number of degree-of-freedom is left unchanged.

This option speed-up the computations of sandwiched thin structures but it can only be used if the shell has an uniform thickness of MEs, i.e. a well defined top and bottom smooth surface. Here, when setting Dirichlet BCs for the displacement, just values on the top and bottom of the shell are used. When mixing solid-shell elements defining a single layer with standard solid elements, on the lateral boundary some displacement degree-of-freedoms need to be linearly constrained by the top and bottom values.

### Equation SmoothMesh

This equation computes the pseudo displacement field  $\mathbf{u}'$  on  $\Omega_{\text{smooth}}$ , therefore allowing the computation of non-mechanical dof-fields on a strongly deformed mesh. The main idea is to iteratively first compute the true displacement fields on  $\Omega_{\text{mech}}$ , then compute the pseudo displacement field  $\mathbf{u}'$  on  $\Omega_{\text{smooth}}$  with  $\mathbf{u}' = \mathbf{u}$  on  $\overline{\Omega_{\text{mech}}} \cap \overline{\Omega_{\text{smooth}}}$  such that  $\varphi = \mathbf{X} + \mathbf{u}'$  is a smooth and one-to-one map and finally to compute the other dof-fields on  $\varphi(\Omega_{\text{smooth}})$  by enabling the model `UpdatedLagrange`. The governing equation for  $\mathbf{u}'$  is governed by the material parameter `SmoothLaw` allowing the generic formulation of a symmetric residual-stiffness law depending upon  $\nabla \mathbf{u}'$ . To the global residual vector  $R$  and global stiffness matrix  $S$ , we add the terms

$$R_i = \int_{\Omega} r(\nabla \mathbf{u}'_h) : \nabla \mathbf{H}_i \, dV, \quad S_{ij} = \int_{\Omega} s(\nabla \mathbf{u}'_h) : [\nabla \mathbf{H}_i \otimes \nabla \mathbf{H}_j] \, dV, \quad (5.45)$$

with  $\mathbf{u}'_h = \sum_i \mathbf{H}_i u'_i$  the finite element approximation of  $\mathbf{u}'$ ,  $\mathbf{H}_i$  the vector shape functions and  $u'_i$  the degree-of-freedoms of  $\mathbf{u}'$ . The residual and stiffness functions  $r$  and  $s$  are specified by the material parameters `SmoothLaw.R<T2U>` and `SmoothLaw.S<T2U>_<T2U>`. We also assume the global stiffness matrix  $S$  to be symmetric. For a generic function  $f(\nabla \mathbf{u}'_h)$  and  $r = \partial f / \partial \nabla \mathbf{u}'_h$ , the first relation in (5.45) is the derivative of the functional  $J = \int_{\Omega} f(\nabla \mathbf{u}'_h) \, dV$  with respect to the degree-of-freedoms  $u'_i$  and the second term is used for the convergence of the Newton-Raphson algorithm and should correspond to the second derivatives  $S_{ij} = \partial^2 J / \partial u'_i \partial u'_j$  resulting in  $s = \partial^2 f / \partial^2 \nabla \mathbf{u}'_h$ . Independently from the selection of the `UpdatedLagrange` model, the displacement gradient  $\nabla \mathbf{u}'$  and the terms (5.45) are always computed in the reference configuration. However, if the `UpdatedLagrange` model is enabled, all other model parameters of Table 5.4 are evaluated in the deformed configuration.

### Equation Poisson Unipolar[N,P,U,V] Bipolar[UV,NP]

These equations solve the drift-diffusion or van Roosbroeck model for semiconductor devices consisting of a system of three PDEs [12, 16]

$$\begin{aligned} \nabla \cdot (\epsilon \nabla \Phi) &= q_0(n - p - C), \\ -q_0 \partial_t n + \nabla \cdot \mathbf{J}_n &= q_0 R, \quad \mathbf{J}_n = q_0 D_n n_{in} e^{\Phi/V_T} \nabla (e^{-\Phi/V_T} n / n_{in}), \\ q_0 \partial_t p + \nabla \cdot \mathbf{J}_p &= -q_0 R, \quad \mathbf{J}_p = -q_0 D_p p_{ip} e^{-\Phi/V_T} \nabla (e^{\Phi/V_T} p / p_{ip}). \end{aligned} \quad (5.46)$$

The first equation of (5.46) is the Poisson equation which couples together the electric potential  $\Phi$  and electric field  $\mathbf{E} = -\nabla \Phi$  with the charge density  $\rho = -q_0(n - p - C)$  composed of the electron and hole carrier densities  $n, p$ , the bulk charge  $C$  and with



$q_0$  the elementary charge and  $\epsilon$  the dielectric permittivity. The second and third equations of (5.46) are the current continuity equations describing the flow of the carrier densities  $n, p$  inside the semiconductor with  $\mathbf{J} = \mathbf{J}_n + \mathbf{J}_p$  the total electric current. The recombination  $R$  accounts for the creation and annihilation of electron-hole pairs inside the semiconductor. For a non-degenerate semiconductor, the diffusion coefficients  $D_{n,p}$  are related to the mobility  $\mu_{n,p}$  by Einstein's relation

$$D_{n,p} = V_T \mu_{n,p}, \quad (5.47)$$

with  $V_T = k_B T / q_0$  the thermal voltage,  $k_B$  the Boltzmann constant and  $T$  the temperature assumed to be overall constant. The input parameters  $\epsilon/\epsilon_0, C, R, \mu_{n,p}, V_T$  to (5.46) together with (5.47) are represented by the material parameters `EpsIso`, `Doping`, `Recomb`, `MobilityN`, `MobilityP` and the global parameter `Vt`. To model heterogeneous semiconductors, we allow the intrinsic densities  $n_{in}$  and  $n_{ip}$  to be piecewise constant functions. They are defined by the intrinsic density  $n_i$  and the shift  $\delta E_i$  of the bandgap center or equivalently of the intrinsic Fermi level  $E_i$  by the relations

$$n_{in} = n_i e^{\delta E_i / V_T}, \quad n_{ip} = n_i e^{-\delta E_i / V_T}, \quad (5.48)$$

with  $n_i, \delta E_i$  defined by the material parameters `Ni` and `dEi`. If a non-zero but overall constant value of  $\delta E_i$  is defined for a semiconductor, the solution will be unaffected except for an overall and constant shift of the electric potential  $\Phi$  by  $\delta E_i$ . When defining inhomogeneous values for  $n_{in}$  and  $n_{ip}$ , large gradients in the potential field at the interfaces between different values are to be expected, see [1, 7]. Here a fine mesh around the interface is a requirement in order to compute meaningful solutions. Further, since the intrinsic densities (5.48) are piecewise constant functions so are the carrier densities which are therefore discontinuous at the interfaces. In the present implementation, an inhomogeneous intrinsic density is correctly represented inside the Poisson equation but only represents a crude approximation of the physics for the current continuity equations. For example, at thermodynamic equilibrium no current flows and from the drift-diffusion model we obtain the relations

$$n = n_{in} e^{\Phi / V_T}, \quad p = n_{ip} e^{-\Phi / V_T}. \quad (5.49)$$

If the semiconductor is not degenerate, i.e. the carrier density is low, then the Boltzmann distribution is a good approximation of the Fermi distribution and the solution of the Poisson equation together with (5.49) yields a good approximation for the potential distribution at thermodynamic equilibrium. We can also use the drift-diffusion model to compute solutions for a non-equilibrium state but here the validity of the model becomes questionable. At the interface of two semiconductors with different bandgaps strong carrier-carrier interactions are present and only Monte Carlo methods are generally able to properly model the physics. The drift-diffusion model based on the linearization of the Boltzmann transport equation cannot describe these strong interactions.

The discretization of the drift-diffusion model (5.46) is performed with the mixed finite element method presented in the Section 4.2 *Mixed Finite Elements* and for the present version, only brick-shaped elements can be used here. According to this method two independent approximations for the field variable and the associated current flux are used. Compatibility conditions are then enforced in a weak form



by the algorithm for each finite element. This discretization method is consistent with the one-dimensional Scharfetter-Gummel discretization scheme of the drift-diffusion model for semiconductors. This method first proposed for semiconductor modeling and the two-dimensional case by [6, 2] has been extended in three dimension in [14]. One of the major drawbacks of the mixed finite element formulation is caused by the evaluation of exponentially fitted integrals, for which special integration algorithms have been devised. For non-brick elements the exact evaluation of these integrals is so expensive that only brick elements are available.

The discretization of the drift-diffusion model (5.46) can be done either with the carrier variables  $(\Phi, n, p)$  or with the Slotboom variables  $(\Phi, u, v)$  defined by the one-to-one maps

$$n = n_{in} e^{\Phi/V_T} u, \quad p = n_{ip} e^{-\Phi/V_T} v, \quad (5.50)$$

see [12]. The advantage of the Slotboom variables is that both continuity equations are now self-adjoint, however, these variables have a much larger range of values than the carrier variables. For heterogeneous materials with discontinuous intrinsic densities  $n_{in}, n_{ip}$ , the Slotboom variables  $(u, v)$  are continuous functions, however, this is not the case for the carrier variables  $(n, p)$  and therefore the discretization is performed with respect to the continuous variables  $(n/n_{in}, p/n_{ip})$ .

When the carrier variables are to be preferred to the Slotboom variables and vice versa depends on the problem being modeled and on the type of solutions being computed. For unstationary solutions only the carrier variables together with the coupled algorithm can be used. For stationary solutions both the Slotboom and the carrier variables are available together with the coupled and uncoupled algorithm. However, for the coupled algorithm, the carrier variables perform generally better since the charge term in the Poisson equation is linear in the carrier densities and leads to a better convergence behavior.

## Equation OpticalMode

This equation computes the quasi-TE or quasi-TM modes of an optical 2D dielectric waveguide which are scalar approximations of the Maxwell equations with zero source terms. These modes are exact TE or TM solutions whenever the refractive index is allowed to change only along a direction perpendicular to the wave propagation. Let us first consider TE and TM modes for a dielectric waveguide with  $\mu = \mu_0$ ,  $\epsilon = n_{\text{refr}}^2 \epsilon_0$ ,  $n_{\text{refr}}$  the refractive index,  $\partial_y = \partial_z = 0$  and the  $z$ -axis as the propagation direction, see [18]. For TE modes the non-zero components of the electric and magnetic fields  $\mathbf{E}, \mathbf{H}$  are  $E_y, H_x, H_z$  and for TM modes  $H_y, E_x, E_z$ . For a dynamical analysis in the Fourier space, we use the Ansatz  $(\bullet)(z, t) = (\bullet)e^{i(\omega t - \beta z)}$  where  $\omega = 2\pi c/\lambda$  is a fixed frequency,  $\lambda$  the wavelength and  $\beta$  the (yet unknown) propagation velocity of the mode in the waveguide. If we assume the refractive index is piecewise constant, then from the Maxwell equations with zero source terms we obtain the following exact equations for the field component  $E_y$  and  $H_y$  inside each layer  $i$  of constant refractive index  $n_i$

$$\begin{aligned} \text{TE modes:} \quad & \left[ \nabla^2 + \left( \frac{2\pi n_i}{\lambda} \right)^2 - \beta^2 \right] \mathcal{E}_y(x) = 0, \quad E_y = \mathcal{E}_y(x) e^{i(\omega t - \beta z)}, \\ \text{TM modes:} \quad & \left[ \nabla^2 + \left( \frac{2\pi n_i}{\lambda} \right)^2 - \beta^2 \right] \mathcal{H}_y(x) = 0, \quad H_y = \mathcal{H}_y(x) e^{i(\omega t - \beta z)}. \end{aligned} \quad (5.51)$$

The other components are derived from the relations

$$\begin{aligned} \text{TE modes: } H_x &= -\frac{\beta}{\omega\mu} E_y, & H_z &= \frac{i}{\omega\mu} \frac{\partial E_y}{\partial x}, \\ \text{TM modes: } E_x &= \frac{\beta}{\omega\epsilon} H_y, & E_z &= \frac{i}{\omega\epsilon} \frac{\partial H_y}{\partial x}. \end{aligned} \quad (5.52)$$

The continuity requirements for the tangential component of the fields  $\mathbf{E}$  and  $\mathbf{H}$  at the dielectric interfaces together with (5.52) yield for TE modes the continuity of  $\mathcal{E}_y$ ,  $\partial\mathcal{E}_y/\partial x$  and for TM-modes the continuity of  $\mathcal{H}_y$ ,  $n_{\text{refr}}^{-2}\partial\mathcal{H}_y/\partial x$ . These continuity requirements at the dielectric interfaces are implicitly defined when solving the following Helmholtz problems for the field components  $\mathcal{E}_y$ ,  $\mathcal{H}_y$  on the domain  $\Omega = \{x \in \mathbb{R}\}$

$$\begin{aligned} \text{TE modes: } & \left[ \nabla^2 + \left( \frac{2\pi n_{\text{refr}}}{\lambda} \right)^2 - (\beta)^2 \right] \mathcal{E}_y(x) = 0, \quad x \in \Omega, \\ \text{TM modes: } & \left[ \nabla \frac{1}{n_{\text{refr}}^2} \nabla + \left( \frac{2\pi}{\lambda} \right)^2 - \left( \frac{\beta}{n_{\text{refr}}} \right)^2 \right] \mathcal{H}_y(x) = 0, \quad x \in \Omega. \end{aligned} \quad (5.53)$$

For a piecewise constant refractive index  $n_{\text{refr}} = n_{\text{refr}}(x)$ , the solutions of these Helmholtz problems are exact solutions of the Maxwell equations. There are in general a finite number of bounded solutions, each one representing a single propagating mode.

In practice the refractive index of optical waveguides cannot be invariant along the  $y$ -axis since the optical mode must be limited laterally in some way. If the variations of the refractive index along the  $y$ -axis are small compared with the ones along the  $x$ -axis, then the optical modes will preserve the TE and TM character and the solution of the Helmholtz problems (5.53) with a refractive index  $n_{\text{refr}} = n_{\text{refr}}(x, y)$  will result in approximate solutions of the Maxwell equations. These solutions are called quasi-TE and quasi-TM modes. Now that the modes are limited laterally the amount of energy transported is finite and given by

$$\begin{aligned} \text{TE modes: Energy Flux} &= -\frac{1}{2} \int_{\Omega} E_y H_x^* dV = \frac{\beta}{2\omega\mu_0} \int_{\Omega} \mathcal{E}_y^2 dV, \\ \text{TM modes: Energy Flux} &= \frac{1}{2} \int_{\Omega} H_y E_x^* dV = \frac{\beta}{2\omega\epsilon_0} \int_{\Omega} \frac{\mathcal{H}_y^2}{n_{\text{refr}}^2} dV. \end{aligned} \quad (5.54)$$

Although for the waveguide problem it is natural to seek solutions of the equations (5.53) in the unbounded domain  $\Omega = \mathbb{R}^2$ , *SESES* only allows bounded domains which automatically arises the question of defining suitable boundary conditions. Because for guided modes the field amplitude decays exponentially to zero at infinity, homogeneous Dirichlet boundary conditions are a good approximation but only if the domain is sufficiently large. Homogeneous Neumann boundary conditions are also of interest on axes of symmetry; however these boundary conditions are automatically set when no boundary conditions are defined. The use of other boundary conditions is not supported.

To solve the Helmholtz problem (5.53) for a TE or a TM optical field, the equation `OpticalMode` must be defined and one of the global models `ModeTE` or `ModeTM` be selected. The optical field is computed with the eigen-solver and the solution is proportional to the field  $\mathcal{E}_y$  for a TE mode and to the field  $\mathcal{H}_y$  for a TM mode. The mode is normed so that the energy flux  $\int \Phi^2 dV$  for TE a mode and  $\int (\Phi/n_{\text{refr}})^2 dV$  for a TM mode is  $1\text{W/m}^2$ .

NOTE. The eigen-solver computes eigen-pairs of positive definite matrices. Since the direct discretization of (5.53) does not yield a positive definite matrix, the problems (5.53) are modified to the following ones

$$\begin{aligned} \text{TE modes: } & \left[ \nabla^2 + \left( \frac{2\pi n}{\lambda} \right)^2 - \left( \frac{2\pi n_{\text{film}}}{\lambda} \right)^2 - (\beta')^2 \right] \mathcal{E}_y(\mathbf{x}) = 0, \mathbf{x} \in \Omega, \\ \text{TM modes: } & \left[ \nabla \frac{1}{n_{\text{refr}}} \nabla + \left( \frac{2\pi}{\lambda} \right)^2 - \left( \frac{2\pi n_{\text{film}}}{\lambda n} \right)^2 - \left( \frac{\beta'}{n_{\text{refr}}} \right)^2 \right] \mathcal{H}_y(\mathbf{x}) = 0, \mathbf{x} \in \Omega, \end{aligned} \quad (5.55)$$

where  $n_{\text{film}}$  is a constant with  $n_{\text{film}} \geq n$  on  $\Omega$ . This modification of the Helmholtz problems leads to a shift of the eigenvalues  $\beta = \beta' - 2\pi n_{\text{film}}/\lambda$ , however, the eigenvectors are left unchanged and the discretization of (5.55) now yields positive definite matrices. The global constant  $n_{\text{film}}$  is defined by the global parameter `Nfilm` and the user must ensure the condition  $n_{\text{film}} \geq n$ . Although by choosing a large default value for  $n_{\text{film}}$  the system matrices will be positive definite for all problems of practical interest, for numerical reasons this procedure should be avoided because it slows down the eigen-solver. In general the best convergence is obtained when  $n_{\text{film}}$  is set to the largest value of the refractive index inside the waveguide, i.e. the refractive index of the film layer.

## 5.2 Material laws

This section describes the collection of models, parameters and built-in functions available to define material laws when solving the governing equations of Section 5.1 *Governing Equations*. A default material law is always set for any equation, but other laws can be defined together with the `GlobalSpec` and `MaterialSpec` statements explained on p. 54. This includes the specification of global parameters listed in Table 5.2 together with their default values or of material parameters listed in Table 5.3. These parameters have been presented together with the associated governing equations in the previous section. The global parameters can only be defined as constants, but many material parameters can be defined as functions of the built-in symbols of `ClassModel` listed in Table 3.10 and Table 5.4. If a material parameter is not constant and if user defined derivatives for the parameter are available, then one has to specify them correctly in order to obtain the best convergence behavior. The default values of the material parameters are just defined to avoid numerical instabilities when solving the governing equations and therefore they need to be properly defined whenever they are used. In this default setting, not a single coupling mechanism is active. If a material law makes a reference to a dof-field not defined by the user with an associated equation, then its value will be zero, except for the temperature  $T$  that will be set according to

$$T(\mathbf{x}) = T_{\text{amb}}, \quad (5.56)$$

with  $T_{\text{amb}}$  the value of the global parameter `AmbientTemp`. This allows to easily perform isothermal simulations without the need to define a constant temperature dof-field.

Non-scalar input values are identified by a dot notation and for values which are physical tensors, the component notation of Table 3.1 is used. Although the tensor notation

Symbol	SESES name	Unit	Value	Description
$T_{\text{amb}}$	AmbientTemp	K	300	Ambient temperature (5.56)
$\omega$	Frequency	1/s	1	Harmonic frequency p. 121
$\alpha$	FlowPartInt		0	Partial integration switcher (5.5)
$V_T$	Vt	V	0.025 V	Thermal voltage (5.46)
$\lambda$	WaveLength	m	$1.3 \times 10^{-6}$	Optical wavelength (5.53)
$n_{\text{film}}$	Nfilm		4	Optical mode shift (5.55)
	PhiEqScaling		1	Equation scaling factor for (5.21)
	MagnPotEqScaling		1	Equation scaling factor for (5.24)
	iAfieldEqScaling		1	Equation scaling factor for (5.36), (5.37), (5.38),

Table 5.2: The global parameters defined with the GlobalSpec Parameter statement and their default values.

allows the definition of general anisotropic material laws, for convenience and numerical optimization, sometimes it is possible to define material laws both in a anisotropic or isotropic form and if both forms are used the last declared one is employed. For the 2D version, the  $z$ -axis is normal to the 2D domain and some tensors may have non-zero components along this axis as well.

Symbol	SESES name	Unit	Description
$\kappa$	KappaIso.Val(MODEL)	W/m/K	Isotropic thermal conductivity (5.17)
$\partial_{...}\kappa$	KappaIso.D<MODEL>	<...>	Generic derivative of $\kappa$
$\kappa$	KappaAni.<T2>(MODEL)	W/m/K	Anisotropic thermal conductivity (5.17)
$\partial_{...}\kappa$	KappaAni.<T2>_D<MODEL>	<...>	Generic derivative of $\kappa$
$\sum_{\alpha} \partial_T h_{\alpha}$ $\leftrightarrow (\rho_{\alpha} \mathbf{v} + \mathbf{j}_{\alpha})$	ThermConv(MODEL)	W/(m*K)	Thermal convective flux (5.18)
$h_{\alpha}$	Enthalpy.< $\alpha$ >(MODEL)	J/kg	Species enthalpy (5.18)
$c_{P,\alpha}$	Enthalpy.< $\alpha$ >_DTemp	J/kg/K	Derivative of $h_{\alpha}$
$q$	Heat.Val(MODEL)	W/m**3	Heat source (5.18)
$\partial_{...}q$	Heat.D<MODEL>	<...>	Generic derivative of $q$
$(\rho c_P)_0$	CpDensity0	J/K/m**3	Solid matter specific heat (5.18)
$\rho_0$	Density0	kg/m**3	Solid matter density (5.40), (5.43)
$\rho$	Density.Val(Temp, Pressure, < $\alpha$ >, MoleMass)	kg/m**3	Material density (5.1), (5.3)
$\partial_T \rho$	Density.DTemp	kg/m**3/K	Derivative of $\rho$
$\partial_p \rho$	Density.DPressure	kg/J	Derivative of $\rho$
$\partial_{x_{\alpha}} \rho$	Density.D< $\alpha$ >	kg/m**3	Derivative of $\rho$
$\mathbf{f}$	Force.<T1>(MODEL)	N/m**3	Body force (5.3), (5.40)
$\partial_{...}\mathbf{f}$	Force.<T1>_D<MODEL>	<...>	Generic derivative of $\mathbf{f}$
$\mu$	Viscosity(MODEL)	kg/m/s	Viscosity (5.4)
$\gamma$	NSConvFac		Convection scaling factor (5.3)
$\delta_a$	FlowStab.A		Stability parameter (5.5)
$\delta_b$	FlowStab.B		Stability parameter (5.5)

$\partial_v \delta_a$	FlowStab. A_DVelocity<T1>		Derivatives of $\delta_a$
$\partial_v \delta_b$	FlowStab. B_DVelocity<T1>		Derivatives of $\delta_b$
$\delta_3$	PressPenalty	s	Stability parameter (5.6)
$k$	Permeability(MODEL)	m**2	Permeability (5.8)
$D_{\alpha\beta}$	Diff.< $\alpha$ >,< $\beta$ >(MODEL)	kg/s/m	Diffusion coefficients (5.11)
$\partial_{x_\beta} j_\alpha$	Diff.<T1>< $\alpha$ >_D< $\beta$ >	kg/s/m**2	Sum derivatives of $D_{\alpha\beta}$
$M_\alpha$	Mmol.< $\alpha$ >	kg/mol	Species molar mass (5.71)
$\nabla(x_\alpha M_\alpha/M) \cdot \rho \mathbf{v}$	TransConv(MODEL)		Transport convective flux (5.14)
$\rho \mathbf{v}$	TransConv.Vel	kg/s/m**2	Mass flow
$M_\alpha/M \delta_{\alpha\beta} - \leftrightarrow \sum_\beta M_\beta/M$	TransConv. < $\alpha$ >_D< $\beta$ >Grad		Derivative
$\partial_{x_\beta} [\nabla(x_\alpha M_\alpha/M) \cdot \rho \mathbf{v}]$	TransConv.< $\alpha$ >_D< $\beta$ >	kg/s/m**2	Derivative
$\Pi_\alpha$	Rate.< $\alpha$ >(MODEL)	kg/s/m**3	Species mass production (5.10)
$\partial_{...} \Pi_\alpha$	Rate.< $\alpha$ >_D<MODEL>	<...>	Generic derivative of $\Pi_\alpha$
$\Pi_0$	Rate0(MODEL)	kg/s/m**3	Mass production as right-hand side of (5.1)
$\sigma$	SigmaIso(MODEL)	S/m	Isotropic electric conductivity (5.20)
$\partial_T \sigma$	SigmaIsoDTemp(MODEL)	S/m/K	Derivative of $\sigma$
$\sigma$	SigmaAni.<T2>(MODEL)	S/m	Anisotropic electric conductivity (5.20)
$\sigma$	SigmaUns.<T2U>(MODEL)	S/m	Unsymmetric electric conductivity (5.20)
$\epsilon_{\text{rel}}$	EpsIso(MODEL)		Isotropic relative dielectric permittivity (5.22), (5.46)
$\epsilon_{\text{rel}}$	EpsAni.<T2>(MODEL)		Anisotropic relative dielectric permittivity (5.22)
$\varrho$	Charge.Val(MODEL)	1/m**3	Charge distribution (5.21)
$\partial_{...} \varrho$	Charge.D<MODEL>	<...>	Generic derivative of $\varrho$
$\mu_{\text{rel}}$	Mue.Val(MODEL)		Rel. permeability (5.24), (5.34)
$H^{-1} \partial_H \mu_{\text{rel}}$	Mue.DlogH	m**2/A**2	Log. derivative of $\mu_{\text{rel}}$ (5.24)
$B^{-1} \partial_B \mu_{\text{rel}}$	Mue.DlogB	1/T**2	Log. derivative of $\mu_{\text{rel}}$ (5.34)
$\Im \mu_{\text{rel}}$	iMuerel		Imaginary part of $\mu_{\text{rel}}$
$\mathbf{H}_0$	Hfield0.<T1>(MODEL)	A/m	External $\mathbf{H}$ -field (5.24)
$\Im \mathbf{H}_0$	iHfield0.<T1>(MODEL)	A/m	Imaginary part of $\mathbf{H}_0$
$G$	Gen.Val(MODEL)	1/(s*m**3)	Generation rate, (5.19)
$\partial_{...} G$	Gen.D<MODEL>	<...>	Generic derivative of $G$
$\mathbf{J}_{00}$	Current00.<T1Z>(MODEL)	A/m**2	External current (5.20)
$\mathbf{J}_0$	Current0.<T1Z>(MODEL)	A/m**2	External current (5.34)
$\Im \mathbf{J}_0$	iCurrent0.<T1Z>(MODEL)	A/m**2	Imaginary part of $\mathbf{J}_0$
$\mathbf{S}$	StressOrtho. <T2Z>(MODEL)	Pa	Orthotropic stress law (5.40), (5.43)
$\partial_E \mathbf{S}$	StressOrtho. <T2Z>_D<T2Z>	Pa	Selected derivatives of $\mathbf{S}$

$\mathbf{S}$	StressSym. <T2Z>(MODEL)	Pa	Symmetric stress law (5.40), (5.43)
$\partial_{\mathbf{E}}\mathbf{S}$	StressSym. <T2Z>_D<T2Z>	Pa	Selected derivatives of $\mathbf{S}$
$\mathbf{S}$	StressGen. <T2Z>(MODEL)	Pa	Generic stress law (5.40), (5.43)
$\partial_{\mathbf{E}}\mathbf{S}$	StressGen. <T2Z>_D<T2Z>	Pa	Derivative of $\mathbf{S}$
$\mathbf{r}, \mathbf{s}$	SmoothLaw.[R<T2U>, S<T2U>_<T2U>](MODEL)		Residual-stiffness vector law (5.45)
$\mathbf{g}$	PiEl.<T3Z>(MODEL)	C/m**2	Piezoelectric tensor (5.57)
$C$	Doping(x)	1/m**3	Bulk charge (5.46)
$R$	Recomb.Val(MODEL)	1/(s*m**3)	Recombination (5.46)
$\partial_n R$	Recomb.DN(MODEL)	1/s	Derivative of $R$
$\partial_p R$	Recomb.DP(MODEL)	1/s	Derivative of $R$
$n_i$	Ni	1/m**3	Intrinsic carrier concentration (5.48)
$\delta E_i$	dEi	eV	Band shift (5.48)
$\mu_n$	MobilityN(MODEL)	m**2/(V*s)	Electron mobility (5.47)
$\mu_p$	MobilityP(MODEL)	m**2/(V*s)	Hole mobility (5.47)
$n_{\text{refr}}$	RefrIndex(MODEL)		Refractive index (5.53)

Table 5.3: The material parameters defined with the MaterialSpec <mat> Parameter statement. The dependency (MODEL) stands for a function of any built-in symbols of ClassMODEL listed in Table 3.10 and Table 5.4.

Symbol	SESES name	Unit	Description
$T$	Temp	K	Temperature (5.18), (5.17)
$\nabla T$	TempGrad.<T1>	K/m	Gradient of $T$
$\mathbf{F}$	TempDiff.<T1>	W/m**2	Thermal diffusion (5.17)
$p$	Pressure	N/m**2	Pressure (5.3)
$\nabla p$	PressGrad.<T1>	Pa/m	Gradient of $p$
$\mathbf{v}$	Velocity.<T1>	m/s	Fluid velocity (5.3)
$\nabla \mathbf{v}$	VelocityGrad.<T2U>	1/s	Gradient of $\mathbf{v}$
$\boldsymbol{\tau}$	ShearStress.<T2>	Pa	Fluidic shear stress (5.4)
$M$	MoleMass	kg/mol	Average molar mass (5.9)
$\rho \mathbf{v}$	MassFlow.<T1>	kg/s/m**2	Mass flow (5.1)
$x_\alpha$	< $\alpha$ >		Species mole fractions (5.10)
$\nabla x_\alpha$	< $\alpha$ >Grad.<T1>	1/m	Gradient of $x_\alpha$
$\mathbf{j}_\alpha$	< $\alpha$ >Diff.<T1>	kg/s/m**2	Species diffusion (5.11)
$\mathbf{u}$	Disp.<T1>	m	Displacement (5.40), (5.43)
$\nabla \mathbf{u}$	DispGrad.<T2UZ>		Displacement gradient (5.42), (5.44)
$\mathbf{E}, \boldsymbol{\varepsilon}$	Strain.<T2Z>		Strain tensor (5.42), (5.44)
$\mathbf{s}$	Stress.<T2Z>		Cauchy stress (5.40), (5.43)
$-\mathbf{S}(0)$	StressInitial.<T2Z>	Pa	The negative stress for a zero strain (5.40), (5.43)



$\Psi$	Psi	V	Electric potential (5.20)
$\nabla\Psi$	PsiGrad	V/m	Gradient of $\Psi$
$\Phi$	Phi	V	Electric potential (5.22), (5.46)
$\nabla\Phi$	PhiGrad	V/m	Gradient of $\Phi$
$\Theta$	MagnPot	A	Magnetic potential (5.24)
$\Im\Theta$	iMagnPot	A	Imaginary part of $\Theta$
$\nabla\Theta$	MagnPotGrad	A/m	Gradient of $\Theta$
$\Im\nabla\Theta$	iMagnPotGrad	A/m	Imaginary part of $\nabla\Theta$
$\mathbf{A}$	Afield.<T1>	V*s/m	$\mathbf{A}$ -field (5.34)
$\Im\mathbf{A}$	iAfield.<T1>	V*s/m	Imaginary part of $\mathbf{A}$
$\overline{\Psi} = \int \Psi dt$	Vint	V*s	Time integral of $\Psi$ (5.34)
$\Im\Psi = \int \Im\Psi dt$	iVint	V*s	Imaginary part of $\overline{\Psi}$
$\mathbf{D}$	Dfield.<T1>	C/m**2	Dielectric displacement (5.22), (5.46)
$\mathbf{E}$	Efield.<T1>	V/m	Electric field (5.20), (5.22), (5.34), (5.46)
$\Im\mathbf{E}$	iEfield.<T1>	V/m	Imaginary part of $\mathbf{E}$
$\mathbf{J} = \boldsymbol{\sigma} \cdot \mathbf{E} + \mathbf{J}_0$	Current.<T1>	A/m**2	Electric current (5.20), (5.46), (5.31) without the term $\nabla \times \mathbf{H}_0$
$\Im\mathbf{J}$	iCurrent.<T1>	A/m**2	Imaginary part of $\mathbf{J}$
$\mathbf{B}$	Bfield.<T1>	T	Magnetic induction $\mathbf{B}$ (5.24), (5.34)
$\Im\mathbf{B}$	iBfield.<T1>	T	Imaginary part of $\mathbf{B}$
$\mathbf{H}$	Hfield.<T1>	A/m	$\mathbf{H}$ -field (5.24), (5.34)
$\Im\mathbf{H}$	iHfield.<T1>	A/m	Imaginary part of $\mathbf{H}$
$n$	DensityN	1/m**3	Electron density (5.46)
$p$	DensityP	1/m**3	Hole density (5.46)
$\mathbf{J}_n$	NCurrent.<T1>	A/m**2	Electron current (5.46)
$\mathbf{J}_p$	PCurrent.<T1>	A/m**2	Hole current (5.46)
$\mathcal{E}_y, \mathcal{H}_y$	Mode		TE or TM optical mode (5.53)

Table 5.4: The additional built-in symbols of ClassMODEL for Table 3.10 available when defining the material parameters of Table 5.3.

## Model PiezoElectric

With model PiezoElectric, the behavior of piezoelectric materials can be simulated. For small deformations, this interaction is expressed by the equations [13]

$$\begin{cases} \mathbf{s} &= \mathbf{C} \cdot \boldsymbol{\varepsilon} - \mathbf{g}^T \cdot \mathbf{E}, \\ \mathbf{D} &= \mathbf{g} \cdot \boldsymbol{\varepsilon} + \boldsymbol{\epsilon} \cdot \mathbf{E}, \end{cases} \quad (5.57)$$

with  $\mathbf{g}$  the piezoelectric tensor of rank 3 defined by the material parameter `PiEl`. This model can be derived by considering the internal energy of the thermodynamical



system and its variation

$$\delta U = \mathbf{E} \cdot \delta \mathbf{D} + \mathbf{s} \cdot \delta \boldsymbol{\varepsilon}. \quad (5.58)$$

By selecting  $\mathbf{E}$  and  $\boldsymbol{\varepsilon}$  as independent variables and applying the Legendre transformation  $\tilde{H} = U - \mathbf{E} \cdot \mathbf{D}$ , we obtain the variation of the electric enthalpy

$$\delta \tilde{H} = -\mathbf{D} \cdot \delta \mathbf{E} + \mathbf{s} \cdot \delta \boldsymbol{\varepsilon}. \quad (5.59)$$

Assuming a linear dependency of  $\mathbf{D}$  and  $\mathbf{s}$  from the independent variables  $\mathbf{E}$ ,  $\boldsymbol{\varepsilon}$ , the law (5.57) is obtained by considering the material constants as second order derivatives of the electric enthalpy  $\tilde{H}$  resulting in the symmetry of the piezoelectric tensor

$$d_{ilm}C_{lmjk} = \frac{\delta D_i}{\delta e_{jk}} = -\frac{\delta^2 \tilde{H}}{\delta e_{jk} \delta E_i} = -\frac{\delta^2 \tilde{H}}{\delta E_i \delta e_{jk}} = -\frac{\delta s_{jk}}{\delta E_i}. \quad (5.60)$$

At the present time, this model only works together with the linearized elasticity model (5.43).

### Model UpdatedLagrange

With this model enabled, the last computed mechanical deformations  $\varphi$  are mapped to the element coordinates so that all element computations are done in the deformed configuration  $\Omega^\varphi = \varphi(\Omega)$  instead of the reference one  $\Omega$ . When using this option, all material laws and boundary conditions need to be transformed to the deformed configuration  $\Omega^\varphi$  by the user. An exception to this rule is given by the equation SmoothMesh, see p.124, always solved in the reference configuration.

### Model MechNonLin

With this model, we solve the geometric non-linear equations of elasticity (5.40), i.e. we consider the geometrical effects due to large displacements. We use here a total Lagrange formulation, where all numerical computations are done in the reference configuration and where all material laws need to be defined. As alternative one can use the model UpdatedLagrange.

When performing a non-linear mechanical analysis coupled with other fields, for example with the Poisson equation (5.21), the question arises if the equations we solve are also valid for a deformed configuration. For simplicity of exposition let us just consider the case of the electrostatic potential governed by the equations (5.21),(5.22) which are similar to the temperature equations (5.18),(5.17) and to the electric potential equations (5.19)-(5.20). Under a deformed configuration, the physical laws do not change but must be written for the deformed system. Therefore, given a mechanical deformation  $\varphi$ , we have to solve on the deformed domain  $\Omega_{\text{elstat}}^\varphi = \varphi(\Omega_{\text{elstat}})$  the equation

$$-\nabla \cdot \varphi(\boldsymbol{\epsilon}^\varphi \cdot \nabla^\varphi \Phi) = q \varrho^\varphi. \quad (5.61)$$

Similar to the mechanical case, this equation is expressed in term of the unknown displacement  $\varphi$ , therefore multiplying both sides with  $\det \nabla \varphi$  and using the Piola identity

$\partial_i((\nabla\varphi)^{-1}\det\nabla\varphi)_{ij} = 0$ , the equation is conveniently written for the undeformed domain  $\Omega_{\text{elstat}}$  in divergence form as

$$-\nabla \cdot (\det\nabla\varphi ((\nabla\varphi)^{-1}\epsilon^\varphi(\nabla\varphi)^{-T}) \cdot \nabla\Phi) = q \varrho^\varphi \det\nabla\varphi. \quad (5.62)$$

The term  $\varrho = \varrho^\varphi \det\nabla\varphi$  is therefore interpreted as the charge density and  $\epsilon = (\det\nabla\varphi)(\nabla\varphi)^{-1}\epsilon^\varphi(\nabla\varphi)^{-T}$  as the dielectric permittivity in the reference configuration, thus obtaining the original equations (5.21)-(5.22). If the model option `UpdatedLagrange` is used, all equations are solved in the deformed domain  $\Omega_{\text{elstat}}^\varphi$  and therefore the user need to specify the tensor  $\epsilon^\varphi$  instead of  $\epsilon$ .

**Attention:** When performing a non-linear mechanical analysis coupled with other fields, except for the Cauchy stress tensor  $S^\varphi$ , all vector and tensor fields are given with respect to the reference configuration. As an example, for the electrostatic potential, the output for the electric and dielectric displacement fields are  $\mathbf{E} = -\nabla\Phi$  and  $\mathbf{D} = \epsilon \cdot \mathbf{E}$  and not the fields  $\mathbf{E}^\varphi = -\nabla^\varphi\Phi$  and  $\mathbf{D}^\varphi = \epsilon^\varphi \cdot \mathbf{E}^\varphi$ .

## Model Coupling1D

---

```
Coupling1D(BlockOffset boff; IOffset ioff; JOffset joff;
NoRow [<dof>]+; NoCol [<dof>]+; [<dof>]+ val[(2*nDof+1)2*nDof] )
```

---

This model allows to couple two disjoint domains over an additional virtual dimension point-to-point. It is mainly used in the 2D version for efficient modeling of planar and thin 3D structures. The whole material defining this model becomes the master domain which is coupled to the slave domain defined by the block offset `BlockOffset` and  $(I, J)$ -ME offsets `IOffset`, `JOffset` locating the slave ME domain with respect to the master material. Master and slave domains cannot overlap and must have the same mesh, but not necessarily the same geometry. The coupling is defined by additional terms to be added to the right-hand-side functions associated with a list of dof-fields and these amendments may depend on the local dof-field values on both sides. To define this coupling, one first define the list of  $n$  dof-fields `<dof>` followed by  $2n(2n + 1)$  numerical values. The first  $n$  values represent the amendments to the rhs of the master domain followed by the  $n$  values for the slave domain. These values typically depend from the dof-fields on the master and slave domain which are accessed by the built-in symbols `Dof0.<dof>` and `Dof1.<dof>`. After the definition of the  $2n$  rhs values, one has to define the  $4n^2$  partial derivatives of these values with respect to the  $2n$  dof-field values. If a block function is used for the definition of the numerical values, see Section 3.5 *Block Functions*, then one can use the names `Val.Res0.<dof>`, `Val.Res1.<dof>`, `Val.Res0.<dof>_DDof0.<dof>`, `Val.Res1.<dof>_DDof1.<dof>`, `Val.Res1.<dof>_DDof0.<dof>`, `Val.Res1.<dof>_DDof1.<dof>` to set residual and partial derivatives. The value of the coupling function on the master and slave domains is not directly available for output and internally its evaluation takes place just for the master domain. However, additionally to the built-in symbols `Dof0.<dof>`, `Dof1.<dof>`, the coupling function has read-write access to user defined element fields, see p. 49, which then allows visualization by copying the numerical value to an element field. From the implementation point of view, the dof-fields on both domains become fully connected in the sense

that for linear systems, convergence is achieved in one single step, however, this additional matrix connectivity increases the complexity of the linear solution step. If the coupling is weak, full connectivity may unnecessarily slow down the solution process and a partial or null connectivity may be a valid alternative. In this case, one can use the options `NoCol [<dof1>]+`, `NoRow [<dof2>]+` to set to zero the rows *<dof1>* and columns *<dof2>* of the dof-field cross-coupling matrix, thus reducing the matrix connectivity.

### Model ZeroCharge

This model turn-off the charge  $\rho = -q_0(n - p - C) = 0$  in the Poisson equation when solving the drift-diffusion model (5.46).

### Global Model Axisymmetric

This option is only available for the 2D version and allows the axi-symmetric analysis of 3D structures. The coordinates  $x, y$  are then interpreted as the  $\rho, z$  values of the cylindrical coordinates  $\rho, \phi, z$ . All references to the variable  $\phi$  have been dropped because of the rotational symmetry. With this model enabled, the user must ensure that the  $x$ -coordinate does not assume negative values.

Differently from a 2D plane strain or plane stress mechanical analysis, where the values of the  $zz$ -stress and  $zz$ -strain components are just used for post-processing and may be as well disregarded, for an axi-symmetric analysis it is important to consider the hoop strain and resulting  $zz$ -stress component [3] which therefore needs to be correctly specified with the material parameters `Stress<...>`. For example this will be the case, if one uses the built-in routines `LinElastIso` explained on page 136.

### Global Model PlaneStress

This option is only available for the 2D version and specifies a plane strain (default) or a plane stress mechanical analysis. For the plane strain case, the  $zz$ -strain component is assumed zero, whereas the  $zz$ -stress component is generally not zero, but it does not enter into the formulation and it is just evaluated for post-processing purposes. For a plane stress analysis, the role of the  $zz$ -strain and  $zz$ -stress is reversed. Since in *SESES* the general mechanical material law defines the stress as a function of the strain, a correct analysis for the plane stress case can be obtained by reversing the role of the  $zz$ -stress and  $zz$ -strain components in the input as well as in the output. This can be accomplished by the user when defining the material law with the material parameters `Stress<...>` by storing inverse elasticity coefficients for the `Stress<...>.ZZ` and `Strain.ZZ` components and by considering `Stress<...>.ZZ` as `Strain.ZZ` and viceversa. If one uses the built-in routines `LinElastIso` explained on p. 136 for the parameter `StressOrtho`, then this change of roles in the elasticity tensor will be automatically taken into consideration. However, the user has to know that now `Stress<...>.ZZ` stands for `Strain.ZZ` and viceversa.

## Global Model **DriftDiffDeflection**

---

```
DriftDiffDeflection(Bfield val.<T1>; MueSca val)
```

---

This model enables the modeling of Galvanometric effects within the drift-diffusion model (5.46) where the deflection of charged carriers in the presence of a magnetic field  $\mathbf{B}$  must be taken into account. For a weak magnetic field in the order of 1 Tesla or less and defined by the parameter `Bfield`, the current relations can be written in the form [15]

$$\mathbf{J}_\alpha = M_\alpha \mathbf{J}_{0,\alpha}, \quad M_\alpha^{-1} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} + \mu_{H,\alpha} \begin{pmatrix} 0 & B_z - B_y \\ -B_z & 0 & B_x \\ B_y - B_x & 0 & 0 \end{pmatrix}, \quad \alpha = n, p, \quad (5.63)$$

with the currents  $\mathbf{J}_{0,\alpha}$  defined by the zero magnetic field drift-diffusion model (5.46). The quantity  $\mu_{H,\alpha}$  is called the Hall mobility and in general differs slightly from the drift-diffusion mobility  $\mu_\alpha = q_0 D_\alpha / (k_B T)$ . It is therefore written in the form  $\mu_{H,\alpha} = \mu_{\text{scat}} \mu_\alpha$  with the scattering value  $\mu_{\text{scat}}$  defined by the parameter `MueSca` (default 1). This model is valid if the Hall angle  $\Theta_H$  defined by  $\tan \Theta_H = \mu_{H,\alpha} |\mathbf{B}|$  will be small which implies  $|\mathbf{B}| < 1$  Tesla.

## Global Model **ModeTE ModeTM**

This model defines if the TE or TM modes of Eq. (5.55) should be computed when solving for the equation `OpticalMode`, default is `ModeTE`.

## Global Model **ShellStore, EasStore**

These models are used together with solid-shell and enhanced assumed strain FEs having hidden internal variables and they allocate additional memory to speed-up the post-solution update of the internal variables. As an example, for the solid-shell elements of p. 123, approximately 8 kBytes of memory are allocated, but assembling is almost twice as fast.

## Built-in function **LinElastIso**

---

```
LinElastIso(Emodule val(MODEL) Pa; PoissonR val; AlphaIso val(MODEL))
```

---

This function is a built-in for the material parameter `StressOrtho` and defines the linear material law

$$\mathbf{S} = \mathbf{C} \cdot (\mathbf{E} - \alpha \mathbf{Id}), \quad (5.64)$$

according to the isotropic elasticity tensor

$$C_{ijkl} = \frac{E_{\text{mod}} \nu}{(1 + \nu)(1 - 2\nu)} \delta_{ij} \delta_{kl} + \frac{E_{\text{mod}}}{2(1 + \nu)} (\delta_{ik} \delta_{jl} + \delta_{il} \delta_{jk}), \quad (5.65)$$

with  $E_{\text{mod}}$  the Young's modulus,  $\nu$  Poisson's ratio defined by the parameters `Emodule` and `PoissonR` with default values of  $E_{\text{mod}} = 1.9 \times 10^{11}$  Pa and  $\nu = 0.09$ . The parameter `Alpha` defines an isotropic initial strain  $\alpha$ , often used to model thermal induced strain. Although such a material law has several drawbacks, one of them being the capability to reach infinite compression with finite energy, it is commonly used and it is valid for large deformations and small strains analysis.

If an analysis other than a 3D analysis is performed, a change of the material law (5.65) is sometimes required and these special conditions are automatically considered by this built-in function. For a 2D the plane strain and a rotational symmetric analysis, the law is essentially the same as (5.65) and one just needs to consider that some strain components are a priori zero. For a 2D plane stress analysis, one has to consider that the  $zz$ -stress component is assumed zero which results in the material law

$$C_{ijkl} = \frac{E_{\text{mod}}\nu}{(1-\nu)^2} \delta_{ij}\delta_{kl} + \frac{E_{\text{mod}}}{2(1+\nu)} (\delta_{ik}\delta_{jl} + \delta_{il}\delta_{jk}), \quad (5.66)$$

where  $zz$ -strain and  $zz$ -stress components have been swapped.

### Built-in function **LinElast**

---

```
LinElast(C val(MODEL).<T4Z> Pa; AlphaIso val(MODEL))
```

---

This function is a built-in for the material parameter `StressSym` and defines the linear material law (5.64) according to the elasticity tensor  $\mathbf{C}$  defined by the parameter `C`. The parameter `Alpha` defines an isotropic initial strain  $\alpha$ , often used to model thermal induced strain. One has to pay attention that the elasticity tensor  $\mathbf{C}$  does not generally corresponds to the physical tensor but it is the tensor required by the numerical analysis. In particular, since for a 2D plane stress analysis, as explained on p. 135, the role of the  $zz$ -strain and  $zz$ -stress components are interchanged, the  $C_{xxzz}$ ,  $C_{yyzz}$ ,  $C_{xyzz}$  coefficients are used to specify the  $zz$ -component of strain according to

$$E_{zz} = C_{zzxx}E_{xx} + C_{zzyy}E_{yy} + 2C_{zzxy}E_{xy}, \quad (5.67)$$

and therefore they are to be interpreted as the *inverse* coefficients.

### Built-in function **ElastLogStrain**

---

```
ElastLogStrain(PlasticEF <element-field>; UseDeformGrad;  
val[9](LogStrain[T1Z],T0[T2Z],T1[T2Z],T2[T2Z],MODEL))
```

---

This function is a built-in for the material parameter `StressSym` combined with the material model `MechNonLin` and allows to define isotropic non-linear elastic laws with respect to log-strain principal values. Let  $\nabla\varphi$  be the deformation gradient,  $\mathbf{C} = \nabla\varphi^T \cdot \nabla\varphi$  the right Cauchy strain tensor and  $\mathbf{C} = \sum_i \lambda_i^2 \mathbf{N}_i \otimes \mathbf{N}_i$  its spectral decomposition. In the case of isotropy, the Cauchy stress  $\mathbf{s}$  has the spectral form  $\mathbf{s} = (\det \nabla\varphi)^{-1} \sum_i \tau_i \mathbf{n}_i \otimes \mathbf{n}_i$  with eigenvectors  $\mathbf{n}_i = \nabla\varphi \cdot \mathbf{N}_i / \lambda_i$  and Kirchhoff stress principal values  $\tau_i$  only depending on the  $\lambda_j$ 's or equivalently on the log-strains  $\varepsilon_j = \log(\lambda_j)$ .

This dependency  $\tau_i(\varepsilon_j)$  together with the derivatives  $\partial\tau_i/\partial\varepsilon_j$  must be returned by the function `val` as  $\tau_0, \tau_1, \tau_2, \partial_{\varepsilon_0}\tau_0, \partial_{\varepsilon_1}\tau_0, \partial_{\varepsilon_2}\tau_0, \partial_{\varepsilon_1}\tau_1, \partial_{\varepsilon_2}\tau_1, \partial_{\varepsilon_2}\tau_2$  and where the log-strains  $\varepsilon_i$  are accessed with the parameter `LogStrain`. The returned values can also be accessed as vector components `S0,S1,S2,C00,C11,C22,C01,C12,C02`. From these values, the routine computes the P2K stress  $\mathbf{S}$  and derivatives  $\partial\mathbf{S}/\partial\mathbf{E}$  with respect to the Green-Lagrange strain  $\mathbf{E} = (\mathbf{C} - \mathbf{Id})/2$ . In addition to the log-strains  $\varepsilon_i$ , the symmetric tensors  $\mathbf{N}_i \otimes \mathbf{N}_i / \lambda_i^2$  can be accessed by the parameters `T0, T1, T2`. The option `PlasticEF` is used to pass an element field of type `<T2Z>` storing a symmetric second order tensor  $\mathbf{C}_0$  used to define an amended strain tensor  $\mathbf{C} = \nabla\varphi^T \cdot (\mathbf{Id} + \mathbf{C}_0) \cdot \nabla\varphi$ . The option `UseDeformGrad` is used together with the `PlasticEF` one for a little speed-up when using FEs where the amended strain can be computed as above from the displacement gradient  $\nabla\varphi$ . This is not always possible as for mixed FEs using internal variables for the representation of the strain. Here an additional square root computation of  $\mathbf{C}_0$  is required.

### Built-in function `PiezoHallRes` `PiezoRes` `HallRes`

---

```
PiezoHallRes(SigmaIso val(MODEL) S/m; PiRes val(MODEL).<T4Z> 1/Pa;
  Bfield val(MODEL).<T1> T; MobHall val(MODEL) m**2/(V*s);
  PiHa val(MODEL).<T4Z> 1/Pa)


---


PiezoRes(SigmaIso val(MODEL) S/m; PiRes val(MODEL).<T4Z> 1/Pa)


---


HallRes(SigmaIso val(MODEL) S/m; Bfield val(MODEL).<T1> T;
  MobHall val(MODEL) m**2/(V*s))


---


```

These functions are a built-ins for the material parameter `SigmaUns` and incorporate the mechanical stress and magnetic field dependency of the resistivity  $\sigma^{-1}$  in a semiconductor material. In general, this dependency is expressed as [15, 9]

$$\sigma^{-1} = \rho_{\text{iso}}(\mathbf{Id} + \mathbf{\Pi}^r \cdot \mathbf{s} - \mu_H \text{Spin}((\mathbf{Id} + \mathbf{\Pi}^h \cdot \mathbf{s})\mathbf{B})), \quad (5.68)$$

with the spin matrix defined by  $\text{Spin}(\mathbf{a})\mathbf{b} = \mathbf{a} \times \mathbf{b}$ ,  $\mathbf{\Pi}^r$  the piezoresistivity tensor defined by the parameter `PiRes`,  $\mathbf{\Pi}^h$  the piezo-Hall tensor defined by the parameter `PiHa`,  $\mathbf{B}$  the magnetic field defined by the parameter `Bfield`,  $\mu_H$  the Hall mobility defined by the parameter `MueHall`,  $\rho_{\text{iso}}$  the isotropic resistivity in absence of mechanical stresses and Hall effects defined by  $\sigma_{\text{iso}}$  and the parameter `SigmaIso` through the relation  $\rho_{\text{iso}} = \sigma_{\text{iso}}^{-1}$  and  $\mathbf{s}$  the computed mechanical stress tensor. This Hall model is only valid if the Hall angle  $\Theta_H$  defined by  $\tan \Theta_H = \mu_H |\mathbf{B}|$  is small which implies  $|\mathbf{B}| < 1$  Tesla [15].

In the case the magnetic field  $\mathbf{B}$  is zero, we obtain the following relation that only describes the piezoresistivity effect

$$\sigma^{-1} = \rho_{\text{iso}}(\mathbf{Id} + \mathbf{\Pi}^r \cdot \mathbf{s}). \quad (5.69)$$

Similarly, if the stress  $\mathbf{s}$  is zero, we obtain the following relation that only describes the Hall effect

$$\sigma^{-1} = \rho_{\text{iso}}(\mathbf{Id} - \mu_H \text{Spin}(\mathbf{B})) = \rho_{\text{iso}}\mathbf{Id} - \rho_{\text{iso}}\mu_H \begin{pmatrix} 0 & -B_z & B_y \\ B_z & 0 & -B_x \\ -B_y & B_x & 0 \end{pmatrix}. \quad (5.70)$$



The general case is considered by the built-in `PiezoHallRes`, whereas the latter two special cases by the built-ins `PiezoRes` and `HallRes`.

### Built-in function `BiotSavart` `BiotSavartEF`

---

```
BiotSavart(Domain val(ME); val(COORD).<T1> A/m**2)
BiotSavartEF(<element-field>)
```

---

These functions are built-ins to compute the Biot-Savart integral (5.25) and require as input the specification of the electrical current  $\mathbf{J}_0$ . In the first form, the current is directly specified as a function of class `ClassCOORD`. The evaluation of the volume integral (5.25) is generally expensive and has to be performed many times, therefore the option `Domain` should be used to define the minimal domain containing the support of the current function. If the current is not given in analytical form but it is computed by *SESES*, you first need to store the current into an element field and then pass the element field as the current function. Here the integrand evaluation is fast and so the computational cost of the integral is dominated by traversing the element mesh and the setting of some internal element data. A good speed-up is obtained by the second form of the built-in, where in addition to the plain current values, the element field provides information on the integration points. This is done by declaring an element field of dimension 6 and by storing there the coordinates of the integration points and the current values multiplied by the volume weight of the integration points so that the sum of all current values corresponds to the integral  $\int_{\mathbb{R}^3} \mathbf{J}_0(\mathbf{y}) d\mathbf{y}$ . In practice, this is done within the `Store` statement explained on p. 86 by storing the value `(coord, Current*VolWeight)` into the element field `<element-field>` passed as parameter to the built-in `BiotSavartEF`.

### Built-in function `BiotSavartPot` `BiotSavartPotEF`

---

```
BiotSavartPot(Domain val(ME); Direction val.<T1>;
val(COORD).<T1> A/m**2)
BiotSavartPotEF(Direction val.<T1>; <element-field>)
```

---

These function are built-ins to compute the integral (5.29) yielding the Biot-Savart's potential  $\Phi_{BS}$  of the field  $\mathbf{H}_0$  for domains with  $\mathbf{J}_0 = 0$ . This potential is used to set the discontinuity (5.27) for the potential  $\Phi$  at the boundary  $\partial\Omega_{\text{jump}}$  between two domains using respectively the reduced and the total scalar potential when solving for the magnetostatic problem. The function parameters are the same as the built-ins `BiotSavart`, `BiotSavartEF` with the additional parameter `Direction` (default `(1, 0, 0)`) specifying the direction of path integration (5.28).

### Built-in function `MueHFunc` `MueBFunc`

---

```
MueHFunc(val(MODEL) T)
MueBFunc(val(MODEL) A/m)
```

---



These functions are built-ins for the material parameter `Mue`, representing the relative permeability  $\mu$  for the magnetostatic model (5.24) or the eddy current model (5.34). For the former case, the permeability may be a function of the  $\mathbf{H}$ -field in which case the logarithmic derivative  $H^{-1}\partial\mu/\partial H$  with  $H = |\mathbf{H}|$  should also be specified. This built-in defines  $\mu$  from the specified  $B - H$  curve `val(MODEL)` whereas the logarithmic derivative is computed by numerical difference. This is handy since the  $B - H$  curve is mostly given in tabulated form. For the other case, the permeability may be a function of the induction  $\mathbf{B}$ -field and the built-in works similarly.

### Built-in function `IdealGas`

---

```
IdealGas(AmbientPress val)
```

---

This function is a built-in for the material parameter `Density` defining the density  $\rho$  of an ideal gas according to

$$\rho = \frac{(P_{\text{amb}} + P)}{RT} \sum_{\alpha} M_{\alpha} x_{\alpha} = \frac{M(P_{\text{amb}} + P)}{RT}, \quad (5.71)$$

with  $p_{\text{amb}}$  the ambient pressure defined by `AmbientPress` and  $M = \sum_{\alpha} M_{\alpha} x_{\alpha}$  the average molar mass with the species molar masses  $M_{\alpha}$  defined by the material parameters `Mmol.<alpha>`. The sum over the species in (5.71) is performed for all locally defined species.

### Built-in function `Radiation`

---

```
Radiation(Bound [<bound>]+; Domain val(ME))
```

---

This built-in function defines incoming radiation from an emitting surface  $S_2 = \partial\Omega_2$  defined by a BC surface `Bound` and it is a built-in function of class `ClassMODELRO` to be used together with Neumann BCs for the temperature. The overall emissive power in all directions of the radiating surface  $S_2$  can be given as  $\epsilon_e \sigma T^4$  with  $T$  the surface temperature,  $\epsilon_e$  the emissivity of the surface and  $\sigma_{\text{sb}} = 5.6704 \times 10^{-8} \text{ W}/(\text{m}^2 \text{K}^4)$  the Stefan-Boltzmann constant. For a black body radiation, the emissive power per solid angle is  $\epsilon_e \sigma_{\text{sb}} T^4 \cos(\theta)/\pi$  with  $\theta$  the angle measured from the normal to the surface. The received thermal flux  $\mathbf{F}$  on a surface  $S_1$  at  $\mathbf{x}_1$  from the radiating surface  $S_2$  is then given by the integral

$$\mathbf{F}(\mathbf{x}_1) \cdot \mathbf{n}_1 = \int_{S_2} \epsilon_e \sigma_{\text{sb}} T^4 \frac{(\mathbf{n}_1 \cdot \mathbf{d})(\mathbf{n}_2 \cdot \mathbf{d})}{\pi |\mathbf{d}|^4} dS, \quad (5.72)$$

with  $\mathbf{d} = \mathbf{x}_2 - \mathbf{x}_1$  the vector joining the emitting and receiving point and  $\mathbf{n}_{1,2}$  the outwards normals to the surfaces  $S_{1,2}$ . The built-in function `Radiation` implements the above surface integral with  $\epsilon_e = 1$  and takes the following parameters. The domain  $\Omega_2$  is defined with the option `Domain` and it is necessary in order to determine the radiation direction of the emitting surface i.e. to specify the normal  $\mathbf{n}_2$ . Contributions from

the above integral are only considered when  $(\mathbf{n}_1 \cdot \mathbf{d}) > 0$ ,  $(\mathbf{n}_2 \cdot \mathbf{d}) < 0$  and the domain  $\Omega_2$  is defined on the inwards of the normal  $\mathbf{n}_2$ . At the present time, the non-linear and non-local coupling introduced by this BC is not considered so that the convergence behavior of Newton's algorithm is only linear.

### Built-in function **StefanMaxwellDiff**

---

```
StefanMaxwellDiff(ZeroEntry <spec>; <α>_<β> val m**2/s)
```

---

This function is a built-in for the material parameter `Diff` and defines the Stefan-Maxwell composition dependency of the diffusion coefficients. For fluids with more than two components, the diffusion coefficients  $D_{\alpha\beta}$  in (5.11) can no longer be expressed by simple formulas as they are implicitly determined by the Stefan-Maxwell equations [5]

$$\nabla x_\alpha = \sum_\beta \frac{M}{\rho \tilde{D}_{\alpha\beta}} \left( \frac{x_\alpha \mathbf{j}_\beta}{M_\beta} - \frac{x_\beta \mathbf{j}_\alpha}{M_\alpha} \right), \quad (5.73)$$

with  $\tilde{D}_{\alpha\beta}$  the binary diffusion coefficients defined by the function parameters  $\langle \alpha \rangle\_ \langle \beta \rangle$  with  $\beta > \alpha$ . By solving the singular system (5.73) for the currents  $\mathbf{j}_\alpha$  and rearranging the result in the form of (5.11), we obtain the diffusion coefficients  $D_{\alpha\beta}$  as complex non-linear functions of the mole fractions  $x_\alpha$ .

Because the diffusion currents  $\mathbf{j}_\alpha$  are defined with respect the center of mass system, we have  $\sum_\alpha \mathbf{j}_\alpha = 0$  and the equations (5.73) can be uniquely inverted by considering the conditions  $\sum_\alpha \mathbf{j}_\alpha = 0$  and  $\sum_\alpha \nabla x_\alpha = \nabla 1 = 0$ . The diffusion coefficients  $D_{\alpha\beta}$  are uniquely defined by choosing the additional conditions  $D_{\alpha\alpha} = 1$  and they fulfill the property  $\sum_\alpha D_{\alpha\beta} = C$ . If the option `ZeroEntry` is enabled, we use the conditions  $D_{\alpha\langle \text{spec} \rangle} = 0$  to uniquely define the diffusion matrix. This zero column in the diffusion matrix is used whenever we use the relation  $x_{\langle \text{spec} \rangle} = 1 - \sum_{\alpha, \alpha \neq \langle \text{spec} \rangle} x_\alpha$  to determine  $x_{\langle \text{spec} \rangle}$  instead of directly solving for the redundant species  $\langle \text{spec} \rangle$ .

### Built-in function **RigidIntersection**

---

```
RigidIntersection[CONTACT](point(...).<T1>)
```

---

This function can be called at any time to detect if the point `point` lies within the rigid-body declared by the routine `RigidBody` with the `Misc` statement, see Table 3.27. The parameter dependency of `point(...)` is determined by the calling environment. If the point lies inside or close to the rigid-body, the point can be uniquely projected to the rigid-body surface along its normal. In this case, the projected point `Px`, `Py`, `Pz`, the surface normal `Nx`, `Ny`, `Nz`, the distance to the surface along the normal `Distance`, two surface orthonormal tangent vectors `T0x`, `T0y`, `T0z` and `T1x`, `T1y`, `T1z`, the surface curvatures along the two tangent vectors `K0`, `K1` are returned in the data structure `CONTACT.<...>`. For a positive/negative distance value `CONTACT.Distance`, the point lies outside/inside the rigid-body. If the point is not close or inside the rigid-body, `CONTACT.Distance` is set to a very large positive value and

the other data of `CONTACT` has no meaning. The `Closeness` parameter of the routine `RigidBody` determines how close must be a point in order to be reported. The rigid-body surface defined by the ME's nodal points is interpolated to be  $C^1$ -smooth so that normal and tangent vectors are continuous functions of the coordinates. To easily interface this built-in with the `Glide` built-in presented on p.142, we also set the value `CONTACT.Alpha` as  $\alpha = \mathbf{N} \cdot \mathbf{P}$ .

## 5.3 Boundary Conditions

Boundary conditions (BC) are an integral part of a model described by partial differential equations and they represent the interaction of the system with the external world. BCs must be specified for the equation to be solved with the BC statement explained on p. 57. Dirichlet BCs are used to set the value of the dof-fields and Neumann BCs are used to set the value of the associated flux-field as defined by Table 5.1. The difference between the total and the associated flux is not influenced by the setting of Neumann BCs and also is not included when displaying BC characteristics.

Additionally to the BC statement explained on p. 57, there are special BCs representing convenient model-related forms designed to model complex physical conditions.

### BC `MassFlowFromVelocity`

When solving the Navier-Stokes equations with a Dirichlet BC for the velocity, the velocity also determines the mass flow at the same boundary and failure to reflect this mass flow with a correct Neumann BC for the pressure results in an inconsistency. This built-in just avoids the direct and consistent definition of the Neumann BC for the pressure.

### BC `Glide`

---

```
Glide(val(coord[T1],Disp[T1],Force[T1]))[4]
```

---

This BC implements friction-less mechanical contacts with rigid-bodies. These contacts are solely specified by constraining the displacement on some regions of the domain's boundary onto the rigid-body surface. Since the contact region is generally unknown, it will be computed as a by-product when solving the governing non-linear equations. The implementation is based on a nodal approach and an exact algebraic fulfillment of the contact constraints without penalty parameters. For each nodal point on the boundary surface, the function `val` is called with the parameters `coord`, `Disp` and `Force` representing the point coordinates  $\mathbf{X}$ , the displacement  $\mathbf{u}$  and the sum of all forces acting upon the nodal point. These two values are generally used to determine if a boundary point is in contact or not. If the point is not a contact point, one should return zero, otherwise the 4 parameters  $(\mathbf{n}, \alpha)$  specifying the rigid-body tangent plane constraining the movement  $\mathbf{n} \cdot (\mathbf{X} + \mathbf{u}) = \alpha$ . The total force acting upon the contact point is nothing else than the value of the residual equations at that

point. These values strongly depend on the discretization but the force direction is a good indicator if the contact point is in compression or tension with the rigid-body. At the present time, the curvature of the rigid-body at the contact point is not specified, therefore when solving the mechanical governing equations a quadratic convergence behavior close to the solution cannot be guaranteed. This is indeed the case for no contacts, if the user has correctly specified the material law's derivatives.

## BC Infinity

---

`Infinity(<dof>; InftyConst val)`

---

This BC adds on the boundary  $\partial\Omega_{\text{infinity}}$  an approximation for the behavior of the field `<dof>` in the un-discretized region when a domain of infinite extent has been truncated. The boundary condition is superior to using either zero Dirichlet or Neumann boundary conditions on the artificial boundary. It is assumed in the implemented approach that the value of the field is zero at infinity, the truncated region is composed only of free space and the Laplace equation is solved there with a diffusion constant of `InftyConst` which is 1 by default. The user must ensure the outer boundary is convex in shape and that the coordinate origin is located towards the center of the computational domain. Under a good choice these two conditions, the infinite element approach is able to offer a good approximation of the influence of the infinite region on the domain.

In the implemented approach, mapped infinite elements of the type proposed by [4] are used to provide an additional layer of elements around the artificial boundary. These special elements are constructed to extend out to infinity in one direction. The shape functions used to represent the field remain unaltered and special mapping functions are used to map the element to the infinite region. For example, in one dimension the mapping for a quadratic mapped infinite element is given by

$$x(\xi) = (2x_1 - x_2)\hat{M}_1(\xi) + x_2\hat{M}_2(\xi) \quad \text{with } -1 \leq \xi \leq 1, \quad (5.74)$$

where the mapping functions are defined as  $\hat{M}_1(\xi) = -\xi/(1-\xi)$  and  $\hat{M}_2(\xi) = 1+\xi/(1-\xi)$ . Using this mapping, one recovers the desired result that  $x(-1) = x_1$ ,  $x(0) = x_2$  and  $x(1) = \infty$ .

## BC voltage

This BC models the effect of an ideal voltage contact for the drift-diffusion model (5.46) by applying the user defined voltage  $V_{\text{applied}}$  and defining the electron and hole density in thermal equilibrium. The following Dirichelt BCs are set

$$\begin{aligned} \phi &= V_{\text{applied}} + V_{\text{builtin}}, \\ n/n_{in} &= \exp\left(\frac{V_{\text{builtin}}}{V_T}\right), \\ p/n_{ip} &= \exp\left(\frac{-V_{\text{builtin}}}{V_T}\right), \\ u &= \exp\left(\frac{-V_{\text{applied}}}{V_T}\right), \\ v &= \exp\left(\frac{V_{\text{applied}}}{V_T}\right), \end{aligned} \quad (5.75)$$

with  $V_{\text{builtin}} = V_T \text{asinh}(-C/(2n_i)) - dE_i$ .

## 5.4 Tensors and their Representation

For materials exhibiting anisotropic properties, i.e. their response to external fields is orientation dependent, the material properties are described by tensors. All tensors used in *SESES* corresponds to true physical and mathematical tensors, no additional scaling factors are introduced as sometimes done by some authors which destroy the tensor property. In *SESES* tensors of maximal rank 4 are used and are in general specified with respect to the global coordinate system (GCS), i.e. the one used to specify the device geometry within the input file. However, material tensors tend to have their simplest form and therefore their tabulated values in a local coordinate system (LCS) often different from the GCS. If the LCS and GCS are different, one has to transform the tensors according to the rule

$$T_{ij...lm}^{\text{GCS}} = V_{iI} V_{jJ} \cdots V_{lL} V_{mM} T_{IJ...LM}^{\text{LCS}}, \quad (5.76)$$

with  $V$  the rotation matrix defining the coordinates of a GCS vector from the ones of a LCS vector i.e.  $\mathbf{x}_{\text{GCS}} = V \mathbf{x}_{\text{LCS}}$ . This orthogonal matrix can be constructed, for example, by knowing a vector  $\mathbf{a}$  parallel to the LCS  $x$ -axis and a vector  $\mathbf{b}$  coplanar to the LCS  $xy$ -plane both vectors expressed with respect to the GCS system and by using the relation

$$V = \left[ \frac{\mathbf{a}}{\|\mathbf{a}\|}, \frac{\mathbf{a} \times \mathbf{b}}{\|\mathbf{a} \times \mathbf{b}\|} \times \frac{\mathbf{a}}{\|\mathbf{a}\|}, \frac{\mathbf{a} \times \mathbf{b}}{\|\mathbf{a} \times \mathbf{b}\|} \right]. \quad (5.77)$$

To simplify the input operations, several built-in functions are provided to perform the tensor transformations (5.76).

TransformT2 (<a,b>Dir val(...).<T1>; Inv; T val(...).<T2>)
TransformT2Z(<a,b>Dir val(...).<T1>; Inv; T val(...).<T2Z>)(* 2D *)
TransformT2U(<a,b>Dir val(...).<T1>; Inv; T val(...).<T2U>)
TransformT3 (<a,b>Dir val(...).<T1>; Inv; T val(...).<T3>)(* 3D *)
TransformT3Z(<a,b>Dir val(...).<T1>; Inv; T val(...).<T3Z>)(* 2D *)
TransformT4 (<a,b>Dir val(...).<T1>; Inv; T val(...).<T4>)(* 3D *)
TransformT4Z(<a,b>Dir val(...).<T1>; Inv; T val(...).<T4Z>)(* 2D *)

The parameters  $\mathbf{aDir}$ ,  $\mathbf{bDir}$  are the vectors  $\mathbf{a}$ ,  $\mathbf{b}$  defining the rotation matrix (5.77) between the LCS and the GCS systems. These functions can be called anywhere and the parameter dependency of  $\mathbf{aDir}(\dots)$ ,  $\mathbf{bDir}(\dots)$  and  $T(\dots)$  is determined by the calling environment. The parameter *Inv* is a flag and if defined the inverse  $V^{-1} = V^T$  of (5.77) is used instead of  $V$ .

## References

- [1] G. P. AGRAWAL, N. K. DUTTA, *Long-wavelength semiconductor lasers*, Van Nostrand, 1986.

- [2] E. ANDERHEGGEN, ET. AL., *Numerical comparison of Pian's 2D hybrid FE-model with some classical device simulation discretization methods using Sever's test diode*, NASECODE VI Conf. Proc., pp. 441-447, 1989.
- [3] K. J. BATHE, *Finite element procedures*, Prentice Hall-International, 1996.
- [4] P. BETTESS, *Infinite Elements* Penshaw Press, Sunderland, 1992.
- [5] R. B. BIRD, W. E. STEWART, E. N. LIGHTFOOT, *Transport Phenomena*, John-Wiley, 1962.
- [6] F. BREZZI, L. D. MARINI AND P. PIETRA, *Two-dimensional exponential fitting and applications to Drift-Diffusion models*, SIAM J. Numer. Anal., 26, pp. 1343-1355, 1989.
- [7] H. C. JR. CASEY, M. B. PANISH, *Heterostructure lasers*, Academic Press, 1978.
- [8] P. G. CIARLET, *Mathematical elasticity, volume I, Three dimensional elasticity*, North-Holland, 1988.
- [9] B. HÄLG, *Piezo-Hall coefficients of n-type silicon*, J. Appl. Phys., 64(1), p. 276, 1988.
- [10] G. Z. HOLZAPFEL, *Nonlinear solid mechanics*, Wiley, 2000.
- [11] S. KLINKEL, F. GRUTTMANN, W. WAGNER, *A robust non-linear solid shell element based on a mixed variational formulation*, Comput. Methods Appl. Mech. Engrg., 195, pp. 179-201, 2006.
- [12] P. A. MARKOWICH, *The Stationary semiconductor device equations*, Springer-Verlag, 1986.
- [13] J. F. NYE, *Physical properties of crystals*, Clarendon Press, Oxford, 1985.
- [14] G. SARTORIS, *A 3D rectangular mixed finite element method to solve the stationary semiconductor equations*, SIAM J. Sci. Comp., Vol. 19, No. 2, 1998.
- [15] K. SEEGER, *Semiconductor physics*, Springer-Verlag, 1973.
- [16] S. SELBERHERR, *Analysis and simulation of semiconductor devices*, Springer-Verlag, 1984.
- [17] P. LE TALLEC, *Numerical methods for nonlinear three-dimensional elasticity*, Handbook of Numerical Analysis by P. G. Ciarlet and J. L. Lions, Vol. III, North-Holland, 1994.
- [18] A. YARIV, *Quantum Electronics*, John Wiley & Sons 1989.